Intro to LLMs and LangChain module

Having gained a solid understanding of the basic concepts of the LangChain library from the first module, it is now time to explore each component in greater detail through the upcoming modules. We start by introducing the most important part of the pipeline - the underlying models! This module will introduce the capabilities of different types of large language models and chat models in detail. It is followed by walking through a hands-on project to summarize news articles. It is worth noting that as we progress through subsequent modules and gain a deeper understanding of additional features, we will refine and enhance the mentioned project and embark on new projects.

Below is the module outline, along with a brief description of the content covered in each lesson.

- **Quick Intro to Large Language Models**: Introducing models like GPT-3, GPT-4, and ChatGPT while discussing their abilities in few-shot learning and demonstrating practical examples of text summarization and translation. We'll also address potential challenges such as hallucinations, biases, and the context size of LLMs.
- **Building Applications Powered by LLMs with LangChain**: LangChain solves key challenges in building applications powered by Large Language Models (LLMs) and makes them more accessible. The lesson will start by covering the use of prompts in chat applications. We will also showcase examples of prompts from LangChain, such as those used in summarization or question-answering chains, highlighting the ease of prompt reuse and customization.
- **Exploring the World of Language Models:** Fundamental distinctions exist between language models and chat models. Notably, chat models are trained to engage in conversations by considering previous messages. In contrast, language models respond to a prompt and rely solely on the information provided within it to answer a query. You will acquire the skills to define various types of prompts for simple applications using each available variation.
- **Exploring Conversational Capabilities with GPT-4 and ChatGPT:** The ChatGPT application provides several benefits that enable it to hold meaningful conversations. During the lesson, you will learn how to pass previous messages to the model and observe how the model effectively utilizes and references these messages when required.
- **Build a News Articles Summarizer:** This lesson is the first hands-on project of the course. We will set up the environment by installing the required libraries and loading access tokens. Next, we will download the contents of a news article and provide them as input to a ChatGPT instance equipped with the GPT-4 model, which will handle the summarization process efficiently. The lesson further explores different prompts to get desired styles. (e.g. a bullet-point list)
- **Using the Open-Source GPT4All Model Locally:** While proprietary models like the GPT family are powerful choices, it is essential to note that there are limitations and restrictions when utilizing them. We will present an open-source model called GPT4ALL, which can be executed locally on your own system. In this lesson, we will delve into the inner workings of this model and demonstrate its seamless integration with the LangChain library, facilitating its user-friendly implementation.

- **What other models can we use? Popular LLM models compared:** The integration of LangChain with numerous models and services opens up exciting new possibilities. In particular, combining various models and services is effortless, leveraging their respective strengths and addressing their limitations. You will see a comprehensive list of different models and their respective advantages and disadvantages. Please note that each model has its license, which may not necessarily cover specific situations. (e.g. commercial use) This module aims to provide you with a comprehensive understanding of the various models the LangChain library offers. We will explore multiple use cases and determine the optimal approach for each scenario, shedding light on the most suitable strategies. In particular, we will examine the fundamental distinctions between prompting the Large Language Models (LLMs) and their Chat model counterparts. Finally, introducing open-source models empowers individuals to run the models locally, eliminating associated costs and enabling further development on top of them.

Quick Intro to Large Language Models

Introduction

In this lesson, we will explore how large language models learn token distributions and predict the next token, allowing them to generate human-like text that can both amaze and perplex us.

We'll start with a quick introduction to the inner workings of GPT-3 and GPT-4, focusing on their few-shot learning capabilities, emergent abilities, and the scaling laws that drive their success. We will then dive into some easy-to-understand examples of how these models excel in tasks such as text summarization and translation just by providing a few examples without the need for fine-tuning.

But it's not all smooth sailing in the world of LLMs. We will also discuss some of the potential pitfalls, including hallucinations and biases, which can lead to inaccurate or misleading outputs. It's essential to be aware of these limitations when using LLMs in use cases where 100% accuracy is paramount. On the flip side, their creative process can be invaluable in tasks where imagination takes center stage.

We will also touch upon the context size and maximum number of tokens that LLMs can handle, shedding light on the factors that define their performance.

LLMs in general:

LLMs are deep learning models with billions of parameters that excel at a wide range of natural language processing tasks. They can perform tasks like translation, sentiment analysis, and chatbot conversations without being specifically trained for them. LLMs can be used without fine-tuning by employing "prompting" techniques, where a question is presented as a text prompt with examples of similar problems and solutions.

- **Architecture:**

  LLMs typically consist of multiple layers of neural networks, feedforward layers, embedding layers, and attention layers. These layers work together to process input text and generate output predictions.

- **Future implications:**

  While LLMs have the potential to revolutionize various industries, it is important to be aware of their limitations and ethical implications. Businesses and workers should carefully consider the trade-offs and risks associated with using LLMs, and developers should continue refining these models to minimize biases and improve their usefulness in different applications. Throughout the course, we will address certain limitations and offer potential solutions to overcome them.

Maximum number of tokens

In the LangChain library, the LLM context size, or the maximum number of tokens the model can process, is determined by the specific implementation of the LLM. In the case of the OpenAI implementation in LangChain, the maximum number of tokens is defined by the underlying OpenAI model being used. To find the maximum number of tokens for the OpenAI model, refer to the `max_tokens` attribute provided on the OpenAI documentation or API.

For example, if you're using the `GPT-3` model, the maximum number of tokens supported by the model is 2,049. The max tokens for different models depend on the specific version and their variants. (e.g., `davinci`, `curie`, `babbage`, or `ada`) Each version has different limitations, with higher versions typically supporting larger number of tokens.
It is important to ensure that the input text does not exceed the maximum number of tokens supported by the model, as this may result in truncation or errors during processing. To handle this, you can split the input text into smaller chunks and process them separately, making sure that each chunk is within the allowed token limit. You can then combine the results as needed.
Here's an example of how you might handle text that exceeds the maximum token limit for a given LLM in LangChain. Mind that the following code is partly pseudocode. It's not supposed to run, but it should give you the idea of how to handle texts longer than the maximum token limit.

```
from langchain.llms import OpenAI
# Before executing the following code, make sure to have
# your OpenAI key saved in the "OPENAI_API_KEY" environment variable.
# Initialize the LLM
llm = OpenAI(model_name="text-davinci-003")

# Define the input text
input_text = "your_long_input_text"

# Determine the maximum number of tokens from documentation
max_tokens = 4097

# Split the input text into chunks based on the max tokens
text_chunks = split_text_into_chunks(input_text, max_tokens)

# Process each chunk separately
results = []
for chunk in text_chunks:
    result = llm.process(chunk)
    results.append(result)

# Combine the results as needed
final_result = combine_results(results)
```

In this example, `split_text_into_chunks` and `combine_results` are custom functions that you would need to implement based on your specific requirements, and we will cover them in later lessons. The key takeaway is to ensure that the input text does not exceed the maximum number of tokens supported by the model.

Note that splitting into multiple chunks can hurt the coherence of the text.

Tokens Distributions and Predicting the Next Token

Large language models like GPT-3 and GPT-4 are pretrained on vast amounts of text data and learn to predict the next token in a sequence based on the context provided by the previous tokens**. GPT-family models use Causal Language modeling,** which **predicts the next token while only having access to the tokens before it.** This process enables LLMs to generate contextually relevant text.

The following code uses LangChain's `OpenAI` class to load GPT-3's Davinci variation using `text-davinci-003` key to complete the sequence, which results in the answer. Before executing the following code, save your OpenAI key in the "OPENAI_API_KEY" environment variable.
Moreover, remember to install the required packages with the following command:
`pip install langchain==0.0.208 deeplake openai tiktoken`.

```
from langchain.llms import OpenAI

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

text = "What would be a good company name for a company that makes colorful socks?"

print(llm(text))
```

The sample code.

```
Rainbow Socks Co.
```

The output.

Tracking Token Usage

You can use the LangChain library's callback mechanism to track token usage. This is currently implemented only for the OpenAI API:

```
from langchain.llms import OpenAI

from langchain.callbacks import get_openai_callback


llm = OpenAI(model_name="text-davinci-003", n=2, best_of=2)
```

```
with get_openai_callback() as cb:
    result = llm("Tell me a joke")
    print(cb)
```

The sample code.

```
Tokens Used: 46
        Prompt Tokens: 4
        Completion Tokens: 42
Successful Requests: 1
Total Cost (USD): $0.0009199999999999999
```

The output.

The callback will track the tokens used, successful requests, and total cost.

Few-shot learning

Few-shot learning is a remarkable ability that **allows LLMs to learn and generalize from limited examples.** Prompts serve as the input to these models and play a crucial role in achieving this feature. With LangChain, examples can be hard-coded, but dynamically selecting them often proves more powerful, enabling LLMs to adapt and tackle tasks with minimal training data swiftly.

This approach involves using the `FewShotPromptTemplate` class, which takes in a `PromptTemplate` and a list of a few shot examples. The class formats the prompt template with a few shot examples, which helps the language model generate a better response. We can streamline this process by utilizing LangChain's FewShotPromptTemplate to structure the approach:

```
from langchain import PromptTemplate
from langchain import FewShotPromptTemplate


# create our examples
examples = [
    {
        "query": "What's the weather like?",
        "answer": "It's raining cats and dogs, better bring an umbrella!"
    }, {
        "query": "How old are you?",
        "answer": "Age is just a number, but I'm timeless."
    }
]
```

```python
# create an example template
example_template = """
User: {query}
AI: {answer}
"""

# create a prompt example from above template
example_prompt = PromptTemplate(
    input_variables=["query", "answer"],
    template=example_template
)

# now break our previous prompt into a prefix and suffix
# the prefix is our instructions
prefix = """The following are excerpts from conversations with an AI
assistant. The assistant is known for its humor and wit, providing
entertaining and amusing responses to users' questions. Here are some
examples:
"""
# and the suffix our user input and output indicator
suffix = """
User: {query}
AI: """

# now create the few-shot prompt template
few_shot_prompt_template = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    prefix=prefix,
    suffix=suffix,
    input_variables=["query"],
    example_separator="\n\n"
)
```

After creating a template, we pass the example and user query, and we get the results

```python
from langchain.chat_models import ChatOpenAI
from langchain import LLMChain

# load the model
chat = ChatOpenAI(model_name="gpt-4", temperature=0.0)

chain = LLMChain(llm=chat, prompt=few_shot_prompt_template)
chain.run("What's the meaning of life?")
```

The sample code

```
To live life to the fullest and enjoy the journey!
```

The output.

Emergent abilities, Scaling laws, and hallucinations

Another aspect of LLMs is their **emergent abilities**, which arise as a result of extensive pre-training on vast datasets. These capabilities are not explicitly programmed but emerge as the model discerns patterns within the data. LangChain models capitalize on these emergent abilities by working with various types of models, such as **chat models and text embedding models**. This allows LLMs to perform diverse tasks, from answering questions to generating text and offering recommendations.

Lastly, **scaling laws** describe the relationship between model size, training data, and performance. Generally, as the model size and training data volume increase, so does the model's performance. However, this improvement is subject to diminishing returns and may not follow a linear pattern. It is essential to weigh the trade-off between model size, training data, performance, and resources spent on training when selecting and fine-tuning LLMs for specific tasks.

While Large Language Models boast remarkable capabilities but are not without shortcomings, one notable limitation is the **occurrence of hallucinations**, in which these models produce text that appears plausible on the surface but is actually factually incorrect or unrelated to the given input. Additionally, LLMs **may exhibit biases** originating from their training data, resulting in outputs that can perpetuate stereotypes or generate undesired outcomes.

Examples with Easy Prompts:

Text Summarization, Text Translation, and Question Answering

In the realm of natural language processing, Large Language Models have become a popular tool for tackling various text-based tasks. These models can be promoted in different ways to produce a range of results, depending on the desired outcome.

Setting Up the Environment

To begin, we will need to install the `huggingface_hub` library in addition to previously installed packages and dependencies. Also, keep in mind to create the Huggingface API Key by navigating to Access Tokens page under the account's Settings. The key must be set as an environment variable with `HUGGINGFACEHUB_API_TOKEN` key.

```
!pip install -q huggingface_hub
```

Creating a Question-Answering Prompt Template

Let's create a simple question-answering prompt template using LangChain

```
from langchain import PromptTemplate

template = """Question: {question}


Answer: """

prompt = PromptTemplate(

        template=template,

    input_variables=['question']

)

# user question

question = "What is the capital city of France?"
```

Next, we will use the Hugging Face model `google/flan-t5-large` to answer the question. The `HuggingfaceHub` class will connect to Hugging Face's inference API and load the specified model.

```
from langchain import HuggingFaceHub, LLMChain

# initialize Hub LLM

hub_llm = HuggingFaceHub(

        repo_id='google/flan-t5-large',

    model_kwargs={'temperature':0}

)
```

```
# create prompt template > LLM chain
llm_chain = LLMChain(
    prompt=prompt,
    llm=hub_llm
)

# ask the user question about the capital of France
print(llm_chain.run(question))
```

The sample code.

```
paris
```

The output.

We can also modify the **prompt template** to include multiple questions.

Asking Multiple Questions

To ask multiple questions, we can either iterate through all questions one at a time or place all questions into a single prompt for more advanced LLMs. Let's start with the first approach:

```
qa = [
    {'question': "What is the capital city of France?"},
    {'question': "What is the largest mammal on Earth?"},
    {'question': "Which gas is most abundant in Earth's atmosphere?"},
    {'question': "What color is a ripe banana?"}
]
res = llm_chain.generate(qa)
print( res )
```

The sample code.

```
LLMResult(generations=[[Generation(text='paris',              generation_info=None)],
[Generation(text='giraffe',    generation_info=None)],    [Generation(text='nitrogen',
generation_info=None)],       [Generation(text='yellow',      generation_info=None)]],
llm_output=None)
```

The output.

We can modify our prompt template to include **multiple questions to implement a second approach**. The language model will understand that we have multiple questions and answer them sequentially. This method performs best on more capable models.

```
multi_template = """Answer the following questions one at a time.

Questions:
{questions}

Answers:
"""
long_prompt = PromptTemplate(template=multi_template, input_variables=["questions"])

llm_chain = LLMChain(
    prompt=long_prompt,
    llm=llm
)
qs_str = (
    "What is the capital city of France?\n" +
    "What is the largest mammal on Earth?\n" +
    "Which gas is most abundant in Earth's atmosphere?\n" +
                "What color is a ripe banana?\n"
)
llm_chain.run(qs_str)
```

The sample code.

```
1. The capital city of France is Paris.
2. The largest mammal on Earth is the blue whale.
3. The gas that is most abundant in Earth's atmosphere is nitrogen.
4. A ripe banana is yellow.
```

The output.

Text Summarization

Using LangChain, we can create a chain for text summarization. First, we need to set up the necessary imports and an instance of the OpenAI language model:

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
```

Next, we define a prompt template for summarization:

```
summarization_template = "Summarize the following text to one sentence: {text}"

summarization_prompt = PromptTemplate(input_variables=["text"], template=summarization_template)

summarization_chain = LLMChain(llm=llm, prompt=summarization_prompt)
```

To use the summarization chain, simply call the `predict` method with the text to be summarized:

```
text = "LangChain provides many modules that can be used to build language model applications. Modules can be combined to create more complex applications, or be used individually for simple applications. The most basic building block of LangChain is calling an LLM on some input. Let’s walk through a simple example of how to do this. For this purpose, let’s pretend we are building a service that generates a company name based on what the company makes."

summarized_text = summarization_chain.predict(text=text)
```

The sample code.

```
LangChain offers various modules for developing language model applications, which can be used alone for simple applications or combined for more complex ones.
```

The output.

Text Translation

It is one of the great attributes of Large Language models that enables them to perform multiple tasks just by changing the prompt. We use the same `llm` variable as defined before. However, pass a different prompt that asks for translating the query from a `source_language` to the `target_language`.

```
translation_template = "Translate the following text from {source_language} to {target_language}: {text}"

translation_prompt = PromptTemplate(input_variables=["source_language", "target_language", "text"], template=translation_template)

translation_chain = LLMChain(llm=llm, prompt=translation_prompt)
```

To use the translation chain, call the `predict` method with the source language, target language, and text to be translated:

```
source_language = "English"

target_language = "French"

text = "Your text here"

translated_text = translation_chain.predict(source_language=source_language, target_language=target_language, text=text)
```

The sample code.

```
Votre texte ici
```

The output.

You can further explore the LangChain library for more advanced use cases and create custom chains tailored to your requirements.

Conclusion

In conclusion, large language models **(LLMs) such as GPT-3, ChatGPT, and GPT-4** have shown remarkable capabilities in generating human-like text, driven by their few-shot learning and emergent abilities. These models excel in other tasks like text summarization and translation, often without the need for fine-tuning. However, it is crucial to acknowledge the **potential pitfalls, such as hallucinations and biases**, that can result in misleading or inaccurate outputs.

While LLMs can be a powerful creative asset, it is essential to be aware of their limitations and use them cautiously in cases requiring absolute accuracy. Furthermore, understanding the **context size and maximum token limitations** is vital to optimizing LLM performance. As we continue to develop and utilize LLMs, balancing their potential benefits with the need to mitigate risks and address their inherent limitations is imperative.

In the next lesson you'll find a first introduction at developing applications leveraging LangChain.

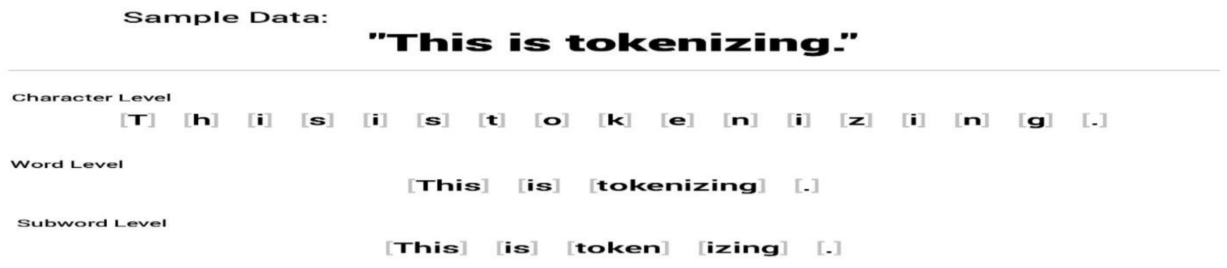
You can find the code of this lesson in this online Notebook.

Understanding Tokens

Tokenization is a fundamental principle in natural language processing (NLP) that plays a crucial role in enabling language models to comprehend written information. It entails **breaking down textual inputs into individual units called tokens**, forming the foundation for effectively understanding and processing language by neural networks. In the previous lesson, we introduced the concept of tokens as a means to define the input for language models (LLMs).

The context length of the model is a frequently discussed characteristic among language models. As an example, the **GPT-3.5 model has a context length of 4096 tokens, covering both the tokens in the prompt and the subsequent completion**. Due to this constraint, it is advisable to be mindful of token usage when making requests to language models. Nonetheless, various approaches exist to tackle this challenge when dealing with long-form inputs or multiple documents. These methods include **breaking up a lengthy prompt into smaller segments and sending requests sequentially, or alternatively, submitting independent requests for each document and merging the responses in a final step.** We will discuss these approaches in more detail throughout the course.
Now, let's delve deeper into what exactly tokens represent and their significance in this context.
The **tokenization process involves creating a systematic pipeline for transforming words into tokens**. However, it is crucial to provide a clear understanding of what exactly tokens represent in this context. Researchers have incorporated three distinct encoding approaches into their studies. The following figure showcases a sample of each approach for your reference.

1. **Character Level**: Consider each character in a text as a token.
2. **Word Level**: Encoding each word in the corpus as one token.
3. **Subword Level**: Breaking down a word into smaller chunks when possible. For example, we can encode the word “basketball” to the combination of two tokens as “basket” + “ball”.


Sample Data:
"This is tokenizing."
Character Level
[T] [h] [i] [s] [i] [s] [t] [o] [k] [e] [n] [i] [z] [i] [n] [g] [.]
Word Level
[This] [is] [tokenizing] [.]
Subword Level
[This] [is] [token] [izing] [.]

The different tokenization process with a sample input. (Photo taken from NLPitation)

Subword-level encoding offers increased flexibility and reduces the number of required unique tokens to represent a corpus. This approach enables the combination of different tokens to represent new words, **eliminating the need to add every new word to the dictionary.** This technique proved to be the most effective encoding when training neural networks and LLMs. Well-known models like **GPT family, LLaMA employ this tokenization method.** Therefore, our sole focus will be on one of its specific variants, known as **Byte Pair Encoding (BPE).** It is worth mentioning that other subword level algorithms exist, such as **WordPiece** and **SentencePiece**, which are used in practice. However, we will not delve into their specifics in this discussion. Nevertheless, it is important to note that while their token selection methods may differ, the fundamental process remains the same.

Byte Pair Encoding (BPE)

It is an iterative process **to extract the most repetitive words or subwords in a corpus**. The algorithm starts by counting the occurrence of each character and builds on top of it by merging the characters. It is a greedy process that carefully considers all possible combinations to identify the optimal set of words/subwords that covers the dataset with the least number of required tokens.

The next step involves creating the vocabulary for our model, which consists of a comprehensive dictionary comprising the most frequently occurring tokens extracted by BPE (or another technique of your choosing) from the dataset. The definition of a dictionary (`dict` type) is a data structure that holds a key and value pair for each row. In our particular scenario, each data point is assigned a **key** represented by an index that begins from 0, while the corresponding **value** is a token.

Due to the fact that **neural networks only accept numerical inputs**, we can utilize the vocabulary to establish a mapping between tokens and their corresponding IDs, like a lookup table. We have to save the vocabulary for future use cases to be able to decode the model's output from the IDs to words. This is known as a **pre-trained vocabulary, an essential component accompanying published pre-trained models.** Without the vocabulary, understanding the model's output (the IDs) would be impossible. For smaller models like **BERT, the dictionary can consist of as few as 30K tokens**, while larger models like **GPT-3 can expand to encompass up to 50K tokens**.

Tokens and Cost

There is a direct correlation between the tokens and the cost in most proprietary APIs like OpenAI. It is crucial to highlight that the prices will fall into two categories: the number of tokens in the prompt and the completion (the model's generation), with the completion typically incurring higher costs. For example, at the time of writing this lesson, GPT-4 will cost $0.03 per 1K tokens for processing your inputs, and $0.06 per 1K tokens for generation process. As we saw in the previous lesson, you can use the `get_openai_callback` method from LangChain to get the exact cost, or do the tokenization process locally and keep track of number of tokens as we see in the next section. For a rough estimation, as a general rule, OpenAI regards 1K tokens to be approximately equal to 750 words.

Tokenizers In Action

In this section, we provide the codes to load the pre-trained tokenizer for the GPT-2 model from the Huggingface Hub using the transformers package. Before running the following codes, you must install the library using the `pip install transformers` command.

```
from transformers import AutoTokenizer

# Download and load the tokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")
```

The code snippet above will grab the tokenizer and load the dictionary, so you can simply use the `tokenizer` variable to encode/decode your text. But before we do that, let's take a look at what the vocabulary contains.

```
print( tokenizer.vocab )
```

The sample code.

```
{'ĠFAC': 44216, 'Ġoptional': 11902, 'Leary': 48487, 'ĠSponsor': 23389, 'Loop': 39516,
'Ġcuc': 38421, 'anton': 23026, 'Ġrise': 4485, 'ĠTransition': 40658, 'Scientists':
29193, 'Ġrehears': 28779, 'ingle': 17697,...
```

The output.
As you can see, each entry is a pair of token and ID. For example, we can represent the word optional with the number 11902. You might have noticed a special character, Ġ, preceding certain tokens. This character represents a space. The next code sample will use the `tokenizer` object to convert a sentence into tokens and IDs.

```
token_ids = tokenizer.encode("This is a sample text to test the tokenizer.")

print( "Tokens:   ", tokenizer.convert_ids_to_tokens( token_ids ) )

print( "Token IDs:", token_ids )
```

The sample code.

```
Output:

Tokens:    ['This', 'Ġis', 'Ġa', 'Ġsample', 'Ġtext', 'Ġto', 'Ġtest', 'Ġthe', 'Ġtoken',
'izer', '.']

Token IDs: [1212, 318, 257, 6291, 2420, 284, 1332, 262, 11241, 7509, 13]
```

The `.encode()` method can convert any given text into a numerical representation, a list of integers. To further investigate the process, we can use the `.convert_ids_to_tokens()` function that shows the extracted tokens. As an example, you can observe that the word "tokenizer" has been split into a combination of "token" + "izer" tokens.

Tokenizers Shortcomings

Several issues with the present tokenization methods are worth mentioning.

- **Uppercase/Lowercase Words**: The tokenizer will treat the the same word differently based on cases. For example, a word like "hello" will result in token id `31373`, while the word "HELLO" will be represented by three tokens as `[13909, 3069, 46]` which translates to `["HE", "LL", "O"]`.
- **Dealing with Numbers**: You might have heard that transformers are not naturally proficient in handling mathematical tasks. One reason for this is the tokenizer's inconsistency in representing each number, leading to unpredictable variations. For instance, the number `200` might be represented as one token, while the number 201 will be represented as two tokens like `[20, 1]`.

- **Trailing whitespace**: The tokenizer will identify some tokens with trailing whitespace. For example a word like “last” could be represented as “ last” as one tokens instead of `[" ", "last"]`. This will impact the probability of predicting the next word if you finish your prompt with a whitespace or not. As evident from the sample output above, you may observe that certain tokens begin with a special character (Ġ) representing whitespace, while others lack this feature.
- **Model-specific**: Even though most language models are using BPE method for tokenization, they still train a new tokenizer for their own models. GPT-4, LLaMA, OpenAssistant, and similar models all develop their separate tokenizers.

Conclusion

We now clearly understand the concept of tokens and their significance in our interaction with language models. You’ve learned the reason why the number of words in a sentence may differ from the number of tokens, as well as how to determine the pricing based on the token count. Nevertheless, the APIs and pipelines available for utilizing LLMs take care of the tokenization process in their backend, relieving you from the burden of handling it yourself. In the upcoming lesson, we will delve into the utilization of LLMs using the LangChain library, where we will explore the concept of chains.

Building Applications Powered by LLMs with LangChain

Introduction

LangChain is designed to assist developers in building end-to-end applications using language models. It offers an array of tools, components, and interfaces that simplify the process of creating applications powered by large language models and chat models. LangChain streamlines managing interactions with LLMs, chaining together multiple components, and integrating additional resources, such as APIs and databases. Having gained a foundational understanding of the library in previous lesson, let's now explore various examples of utilizing prompts to accomplish multiple tasks.

Prompt use case:

A key feature of LangChain is its support for prompts, which encompasses prompt management, prompt optimization, and a generic interface for all LLMs. The framework also provides common utilities for working with LLMs.

`ChatPromptTemplate` is used to create a structured conversation with the AI model, making it easier to manage the flow and content of the conversation. In LangChain, message prompt templates are used to construct and work with prompts, allowing us to exploit the underlying chat model's potential fully.

System and Human prompts differ in their roles and purposes when interacting with chat models. `SystemMessagePromptTemplate` provides initial instructions, context, or data for the AI model, while `HumanMessagePromptTemplate` are messages from the user that the AI model responds to.

To illustrate it, let’s create a chat-based assistant that helps users find information about movies. Ensure your OpenAI key is stored in environment variables using the “OPENAI_API_KEY” name. Remember to install the required packages with the following command: `pip install langchain==0.0.208 deeplake openai tiktoken`.

```python
from langchain.chat_models import ChatOpenAI
from langchain.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)

# Before executing the following code, make sure to have
# your OpenAI key saved in the “OPENAI_API_KEY” environment variable.
chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)

template = "You are an assistant that helps users find information about movies."
system_message_prompt = SystemMessagePromptTemplate.from_template(template)
```

```
human_template = "Find information about the movie {movie_title}."

human_message_prompt = HumanMessagePromptTemplate.from_template(human_template)

chat_prompt = ChatPromptTemplate.from_messages([system_message_prompt, human_message_prompt])

response = chat(chat_prompt.format_prompt(movie_title="Inception").to_messages())

print(response.content)
```

The sample code.

```
Inception is a 2010 science fiction action film directed by Christopher Nolan. The film stars Leonardo DiCaprio, Ken Watanabe, Joseph Gordon-Levitt, Ellen Page, Tom Hardy, Dileep Rao, Cillian Murphy, Tom Berenger, and Michael Caine. The plot follows a professional thief who steals information by infiltrating the subconscious of his targets. He is offered a chance to have his criminal history erased as payment for the implantation of another person's idea into a target's subconscious. The film was a critical and commercial success, grossing over $829 million worldwide and receiving numerous accolades, including four Academy Awards.
```

The output.

Using the `to_messages` object in LangChain allows you to convert the formatted value of a chat prompt template into a list of message objects. This is useful when working with chat models, as it provides a structured way to manage the conversation and ensures that the chat model can understand the context and roles of the messages.

Summarization chain example:

LangChain prompts can be found in various use cases, such as summarization or question-answering chains. For example, when creating a **summarization chain**, LangChain enables interaction with an external data source to fetch data for use in the generation step. This could involve summarizing a lengthy piece of text or answering questions using specific data sources.
The following code will initialize the language model using `OpenAI` class with a temperature of 0 - because we want deterministic output.The `load_summarize_chain` function accepts an instance of the language model and returns a pre-built summarization chain. Lastly, the `PyPDFLoader` class is responsible for loading PDF files and converting them into a format suitable for processing by LangChain.
It is important to note that you need to install the `pypdf` package to run the following code. Although it is highly recommended to install the latest versions of this package, the codes have been tested on version `3.10.0`. Please refer to course introduction lesson for more information on installing packages.

```
# Import necessary modules

from langchain import OpenAI, PromptTemplate

from langchain.chains.summarize import load_summarize_chain

from langchain.document_loaders import PyPDFLoader
```

```
# Initialize language model

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


# Load the summarization chain

summarize_chain = load_summarize_chain(llm)


# Load the document using PyPDFLoader

document_loader = PyPDFLoader(file_path="path/to/your/pdf/file.pdf")

document = document_loader.load()


# Summarize the document

summary = summarize_chain(document)

print(summary['output_text'])
```

The sample code.

```
This document provides a summary of useful Linux commands for starting and stopping, accessing and mounting file systems, finding files and text within files, the X Window System, moving, copying, deleting and viewing files, installing software, user administration, little known tips and tricks, configuration files and what they do, file permissions, X shortcuts, printing, and a link to an official Linux pocket protector.
```

The output. *The output above is based on the "The One Page Linux Manual" PDF file accessible at this URL.*

In this example, the code uses the default summarization chain provided by the `load_summarize_chain` function. However, you can customize the summarization process by providing prompt templates.

Let's recap: OpenAI is initialized with a temperature of 0 for focused and deterministic language model generation. The `load_summarize_chain` function loads a summarization chain, and PyPDFLoader fetches PDF data, which is loaded as a string input for the summarization chain, generating a summary of the text.

QA chain example:

We can also use LangChain to manage prompts for asking general questions from the LLMs. These models are proficient in addressing fundamental inquiries. Nevertheless, it is crucial to remain mindful of the potential issue of **hallucinations**, where the models may generate non-factual information**. To address this concern**, we will later introduce the **Retrieval chain as a means to overcome this problem**.

```
from langchain.prompts import PromptTemplate

from langchain.chains import LLMChain

from langchain.llms import OpenAI
```

```
prompt = PromptTemplate(template="Question: {question}\nAnswer:",
input_variables=["question"])

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

chain = LLMChain(llm=llm, prompt=prompt)
```

We define a custom prompt template by creating an instance of the `PromptTemplate` class. The template string contains a placeholder `{question}` for the input question, followed by a newline character and the "Answer:" label. The `input_variables` argument is set to the list of available placeholders in the prompt (like a question in this case) to indicate the name of the variable that the chain will replace in the template `.run()` method.
We then instantiate an OpenAI model named `text-davinci-003` with a temperature of 0. The `OpenAI` class is used to create the instance, and the `model_name` and `temperature` arguments are provided. Finally, we create a question-answering chain using the `LLMChain` class.
The class constructor takes two arguments: `llm`, which is the instantiated OpenAI model, and `prompt`, which is the custom prompt template we defined earlier.
By following these steps, we can process input questions effectively with the custom question-answering, generating appropriate answers using the OpenAI model and the custom prompt template.

```
chain.run("what is the meaning of life?")
```

The sample code.

```
'The meaning of life is subjective and can vary from person to person. For some, it
may be to find happiness and fulfillment, while for others it may be to make a
difference in the world. Ultimately, the meaning of life is up to each individual to
decide.’
```

The output.

This example demonstrates how LangChain simplifies the integration of LLMs with custom data sources and prompt templates for question-answering applications. To build more advanced NLP applications, you can further extend this example to include other components, such as data-augmented generation, agents, or memory features.
LangChain's support for **chain sequences** also allows developers to create more complex applications with multiple calls to LLMs or other utilities. These chains can serve various purposes: personal assistants, chatbots, querying tabular data, interacting with APIs, extraction, evaluation, and summarization.

Conclusion

LangChain solves the problem of easy integration with other sources of data, tools, and different LLMs by providing a comprehensive framework for managing prompts, optimizing them, and creating a universal interface for all LLMs.
In the next lesson we’ll learn more about popular language models and the recent trend in chat-based language models.

RESOURCES:

langchain

Building applications with LLMs through composability

A Complete Guide to LangChain: Building Powerful Applications with Large Language Models

LangChain is a powerful framework that simplifies the process of building advanced language model applications.

Summarization | 🦜🔗 Langchain

A summarization chain can be used to summarize multiple documents. One way is to input multiple smaller documents, after they have been divided into chunks, and operate over them with a MapReduceDocumentsChain. You can also choose instead for the chain that does summarization to be a StuffDocumentsChain, or a RefineDocumentsChain.

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

Exploring the World of Language Models: LLMs vs Chat Models

Introduction

Large Language Models have made significant advancements in the field of Natural Language Processing (NLP), enabling AI systems to understand and generate human-like text. ChatGPT is a popular language model based on Transformers architecture, enabling it to understand long texts and figure out how words or ideas are connected. It's great at making predictions about language and relationships between words.

LLMs and Chat Models are two types of models in LangChain, serving different purposes in natural language processing tasks. This lesson will examine the differences between LLMs and Chat Models, their unique use cases, and how they are implemented within LangChain.

Understanding LLMs and Chat Models

LLMs

LLMs, such as **GPT-3, Bloom, PaLM, and Aurora genAI**, take a **text string as input** and **return a text string as output**. They are trained on language modeling tasks and can generate human-like text, perform complex reasoning, and even write code. LLMs are powerful and flexible, capable of generating text for a wide range of tasks. However, they can sometimes produce incorrect or nonsensical answers, and their **API is less structured compared to Chat Models.**

Pre-training these models involves presenting large-scale corpora to them and allowing the network to predict the next word, which results in learning the relationships between words. This learning process enables LLMs to generate high-quality text, which can be applied to an array of applications, such as automatic form-filling and predictive text on smartphones.

Most of these models are trained on general purpose training dataset, while others are trained on a mix of general and domain-specific data, such as Intel Aurora genAI, which is trained on general text, scientific texts, scientific data, and codes related to the domain. The goal of domain specific LLMs is to increase the performance on a particularly domain, while still being able to solve the majority of tasks that general LLM can manage.

LLMs have the potential to infiltrate various aspects of human life, including the arts, sciences, and law. With continued development, LLMs will become increasingly integrated into our educational, personal, and professional lives, making them an essential technology to master.

You can follow these **steps to use a large language model (LLM) like GPT-3 in LangChain.** Import the `OpenAI` wrapper from the `langchain.llms` module and Initialize it with the desired model name and any additional arguments. For example, set a high temperature for more random outputs. Then, create a `PromptTemplate` to format the input for the model. Lastly, define an `LLMChain` to combine the model and prompt. Run the chain with the desired input using `.run()`. As mentioned before, remember to set your OpenAI key saved in the "OPENAI_API_KEY" environment variable before running the following codes.

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

```
from langchain.llms import OpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate

# Before executing the following code, make sure to have
# your OpenAI key saved in the “OPENAI_API_KEY” environment variable.
llm = OpenAI(model_name="text-davinci-003", temperature=0)

prompt = PromptTemplate(
  input_variables=["product"],
  template="What is a good name for a company that makes {product}?",
)

chain = LLMChain(llm=llm, prompt=prompt)

print( chain.run("wireless headphones") )
```

The sample code.

```
Wireless Audio Solutions
```

The output.

Here, the input for the chain is the string "wireless headphones". The chain processes the input and generates a result based on the product name.

Chat Models

Chat Models are the most popular models in LangChain, such **as ChatGPT** that can incorporate GPT-3 or GPT-4 at its core. They have gained significant attention due to their **ability to learn from human feedback and their user-friendly chat interface.**

Chat Models, such as ChatGPT, take a **list of messages as input and return an** `AIMessage`. They typically **use LLMs as their underlying technology**, but their **APIs are more structured.**
Chat Models are designed **to remember previous exchanges** with the user in a session and use that context to generate more relevant responses. They also benefit from **reinforcement learning from human feedback**, which helps improve their responses. However, they may still have **limitations in reasoning** and require careful handling to avoid generating inappropriate content.
API Differences in LangChain

Chat Message Types

In LangChain, three main types of messages are used when interacting with chat models: `SystemMessage`, `HumanMessage`, and `AIMessage`.

SystemMessage: These messages provide initial instructions, context, or data for the AI model. They set the objectives the AI should follow and can help in controlling the AI's behavior. System messages are not user inputs but rather guidelines for the AI to operate within.
HumanMessage: These messages come from the user and represent their input to the AI model. The AI model is expected to respond to these messages. In LangChain, you can customize the human prefix (e.g., "User") in the conversation summary to change how the human input is represented.
AIMessage: These messages are sent from the AI's perspective as it interacts with the human user. They represent the AI's responses to human input. Like HumanMessage, you can also customize the AI prefix (e.g., "AI Assistant" or "AI") in the conversation summary to change how the AI's responses are represented.

An example of using ChatOpenAI with a HumanMessage:
In this section, we are trying to use the LangChain library to create a chatbot that can translate an English sentence into French. This particular use case goes beyond what we covered in the previous lesson. We'll be employing multiple message types to differentiate between users' queries and system instructions instead of relying on a single prompt. This approach will enhance the model's comprehension of the given requirements.

First, we create a list of messages, starting with a `SystemMessage` that sets the context for the chatbot, informing it that its role is to be a helpful assistant translating English to French. We then follow it with a `HumanMessage` containing the user's query, like an English sentence to be translated.

```
from langchain.chat_models import ChatOpenAI
from langchain.schema import (
  HumanMessage,
  SystemMessage
)
chat = ChatOpenAI(model_name="gpt-4", temperature=0)


messages = [
        SystemMessage(content="You are a helpful assistant that translates English to 
French."),
        HumanMessage(content="Translate the following sentence: I love programming.")
]
chat(messages)
```

The sample code.

```
AIMessage(content="J'aime la programmation.", additional_kwargs={}, example=False)
```

The output.

As you can see, we pass the list of messages to the chatbot using the `chat()` function. The chatbot processes the input messages, considers the context provided by the system message, and then translates the given English sentence into French.

`SystemMessage` represents the messages generated by the system that wants to use the model, which could include instructions, notifications, or error messages. These messages are not generated by the human user or the AI chatbot but are instead produced by the underlying system to provide context, guidance, or status updates.

Using the generate method, you can also generate completions for multiple sets of messages. Each batch of messages can have its own `SystemMessage` and will perform independently. The following code shows the first set of messages translate the sentences from English to French, while the second ones do the opposite.

```
batch_messages = [
  [
    SystemMessage(content="You are a helpful assistant that translates English to French."),
    HumanMessage(content="Translate the following sentence: I love programming.")
  ],
  [
    SystemMessage(content="You are a helpful assistant that translates French to English."),
    HumanMessage(content="Translate the following sentence: J'aime la programmation.")
  ],
]
print( chat.generate(batch_messages) )
```

The sample code.

```
LLMResult(generations=[[ChatGeneration(text="J'aime la programmation.", generation_info=None, message=AIMessage(content="J'aime la programmation.", additional_kwargs={}, example=False))], [ChatGeneration(text='I love programming.', generation_info=None, message=AIMessage(content='I love programming.', additional_kwargs={}, example=False))]], llm_output={'token_usage': {'prompt_tokens': 65, 'completion_tokens': 11, 'total_tokens': 76}, 'model_name': 'gpt-4'})
```

The output.

As a comparison, here's **what LLM and Chat Model APIs look like in LangChain.**

```
llm_output:  {'product': 'Translate the following text from English to French: Hello,
how are you?', 'text': '\n\nBonjour, comment allez-vous?'}


chat_output:  content='Bonjour, comment ça va ?' additional_kwargs={} example=False
```

The output**.**

Conclusion

LLMs and Chat Models each have their advantages and disadvantages. LLMs are powerful and flexible, capable of generating text for a wide range of tasks. However, their API is less structured compared to Chat Models.
On the other hand, Chat Models offer a more structured API and are better suited for conversational tasks. Also, they can remember previous exchanges with the user, making them more suitable for engaging in meaningful conversations. Additionally, they benefit from reinforcement learning from human feedback, which helps improve their responses. They still have some limitations in reasoning and may require careful handling to avoid hallucinations and generating inappropriate content.
In the next lesson we'll see how GPT-4 and ChatGPT can be used for context-aware chat applications via APIs.

RESOURCES:

ChatGPT and the Large Language Models (LLMs).
ENGLISH—LEARN ABOUT ChatTGPT TECHNOLOGY AND APPLICATIONS
medium.com

Large language models (LLMs) vs. ChatGPT
ChatGPT is a large language model but not every LLM is ChatGPT. Discover interesting applications, how models are trained, and what this tech means for society.

www.thoughtspot.com

A Complete Guide to LangChain: Building Powerful Applications with Large Language Models
LangChain is a powerful framework that simplifies the process of building advanced language model applications.
notes.replicatecodex.com

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

Exploring Conversational Capabilities with GPT-4 and ChatGPT

Introduction

In this lesson, we will explore the benefits of using GPT-4 and ChatGPT, focusing on their ability to maintain context in conversations. We will demonstrate how these advanced language models can remember conversation history and respond accordingly, making them ideal for chat applications. Additionally, we will briefly discuss the improvements in GPT-4, such as longer context length and better generalization. By the end of this lesson, you should be able to understand how GPT-4 and ChatGPT can be used for context-aware chat applications via the API, as opposed to just using the OpenAI ChatGPT webpage.

As mentioned before, **OpenAI's GPT-4** represents a significant advancement in the field of large language models. Among its many improvements are enhanced creativity, the ability to process visual input, and an extended contextual understanding. In the realm of conversational AI, both **GPT-4 and ChatGPT use the Transformers architecture** at their core and are fine-tuned to hold natural dialogue with a user. While the free version of ChatGPT relies on GPT-3, the premium offering, ChatGPT Plus, gives access to the more advanced GPT-4 model.

The benefits of employing ChatGPT and GPT-4 in chat format are numerous. For instance, **GPT-4's short-term memory capacity of 64,000 words** greatly surpasses GPT-3.5's 8,000-word limit, enabling it to maintain context more effectively in prolonged conversations. Furthermore, **GPT-4 is highly multilingual, accurately handling up to 26 languages,** and boasts improved steering capabilities, allowing users to tailor responses with a custom "personality."

The new model is considerably safer to use, boasting a **40% increase in factual responses and an 82% reduction in disallowed content responses**. It can also interpret images as a foundation for interaction. While this functionality has not yet been incorporated into ChatGPT, its potential to revolutionize context-aware chat applications is immense.

Setting up the API

To use GPT-4 or ChatGPT in your application, you must obtain API keys from OpenAI. You'll need to sign up for an account and submit a request to access the latest model. At the time of writing this lesson, there is a waitlist to get your hands on GPT-4. Then, set the `OPENAI_API_KEY` key in your environment variables so the LangChain library can access them.

The following example demonstrates how to create a chatbot using the GPT-4 model from OpenAI. After importing the necessary classes, we declare a set of messages. It starts by setting the context for the model (`SystemMessage`) that it is an assistant, followed by the user's query (`HumanMessage`), and finishes by defining a sample response from the AI model (`AIMessage`).

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

```python
from langchain.chat_models import ChatOpenAI

from langchain.schema import (

    SystemMessage,

    HumanMessage,

    AIMessage

)


messages = [

    SystemMessage(content="You are a helpful assistant."),

    HumanMessage(content="What is the capital of France?"),

    AIMessage(content="The capital of France is Paris.")

]
```

When the user posed the question about the capital of France, the model confidently answered with "Paris." Next up, we test if the model can leverage these discussions as a reference to delve further into details about the city without us explicitly mentioning the name (referring to Paris). The code below adds a new message which requires the model to understand and find the “city you just mentioned” reference from previous conversations.

```python
prompt = HumanMessage(

    content="I'd like to know more about the city you just mentioned."

)

# add to messages

messages.append(prompt)


llm = ChatOpenAI(model_name="gpt-4")


response = llm(messages)
```

```
AIMessage(content='Paris, the capital of France, is one of the most famous and
visited cities in the world. It is located in the north-central part of the country,
along the Seine River. With a population of over 2 million people, it is a bustling
metropolis known for its art, culture, history, and gastronomy.\n\nParis is often
referred to as the "City of Light" (La Ville Lumière) due to its role in the Age of
Enlightenment and its early adoption of street lighting. The city is home to numerous
iconic landmarks, including the Eiffel Tower, the Louvre Museum, Notre-Dame
Cathedral, and the Arc de Triomphe. \n\nParis is also known for its charming
neighborhoods, such as Montmartre, Le Marais, and Saint-Germain-des-Prés, each with
its own unique character and attractions. The city has an extensive public
transportation system, including buses, trams, and the Métro, making it easy to
explore its various districts.\n\nAs a global center for art, fashion, and culture,
Paris hosts numerous events and exhibitions throughout the year, such as Paris
Fashion Week and the Paris Air Show. The city is also renowned for its culinary
scene, featuring a wide array of restaurants, cafés, patisseries, and food markets
that offer both traditional French cuisine and international flavors.\n\nIn addition
to its historical and cultural attractions, Paris is an important international
business hub and the headquarters for many multinational corporations and
organizations, including UNESCO, the OECD, and the International Chamber of
Commerce.', additional_kwargs={}, example=False)
```

As you can see, the model successfully extracted the information from previous conversations and explained more details about Paris. It shows that the chat models are capable of referring to the chat history and understanding the context.

To recap, the `ChatOpenAI` class is used to create a chat-based application that can handle user inputs and generate responses using the GPT-4 language model. The conversation is initiated with a series of messages, including system, human, and AI messages. The `SystemMessage` provides context for the conversation, while `HumanMessage` and `AIMessage` represent the user and the AI's messages, respectively.

The LangChain's Chat API offers several advantages:

- **Context preservation**: By maintaining a list of messages in the conversation, the API ensures that the context is preserved throughout the interaction. This allows the GPT-4 model to generate relevant and coherent responses based on the provided information.
- **Memory**: The class's message history acts as a short-term memory for the chatbot, allowing it to refer back to previous messages and provide more accurate and contextual responses.
- **Modularity**: The combination of MessageTemplate and ChatOpenAI classes offers a modular approach to designing conversation applications. This makes it easier to develop, maintain, and extend the functionality of the chatbot.
- **Improved performance**: GPT-4, as an advanced language model, is more adept at understanding complex prompts and generating better responses than its predecessors. It can handle tasks that require deeper reasoning and context awareness, which leads to a more engaging and useful conversation experience.
- **Flexibility**: The Chat API can be adapted to different domains and tasks, making it a versatile solution for various chatbot applications. In this example, the chatbot specializes in French culture but could be easily modified to focus on other subjects or industries. Moreover, as newer and more powerful language models become available, the API can be updated to utilize those models, allowing for continuous improvements in chatbot capabilities.

Conclusion

In this lesson, we learned that GPT-4 boasts remarkable advancements in context length and generalization, paving the way for more sophisticated language processing. By accommodating a more extensive context, GPT-4 can generate lengthier text pieces, analyze more massive documents, and engage in longer conversations without compromising the context's integrity.
ChatGPT and GPT-4 models are tailor-made for conversational interfaces, which require input to be formatted in a specific chat-like transcript format. This format empowers the models to retain conversation history and furnish contextually relevant responses. This attribute is especially advantageous for multi-turn conversations and can also prove useful in non-chat scenarios.

These potent models have diverse applications, including customer support chatbots that manage intricate inquiries and dispense pertinent responses based on previous interactions. They can also function as virtual personal assistants that preserve context across various tasks and requests. They also serve as natural language interfaces for databases and search engines, enabling them to better understand user queries and provide more accurate results.
In the next lesson, you'll do the first project of the course, that is a news summarizer leveraging LangChain.

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

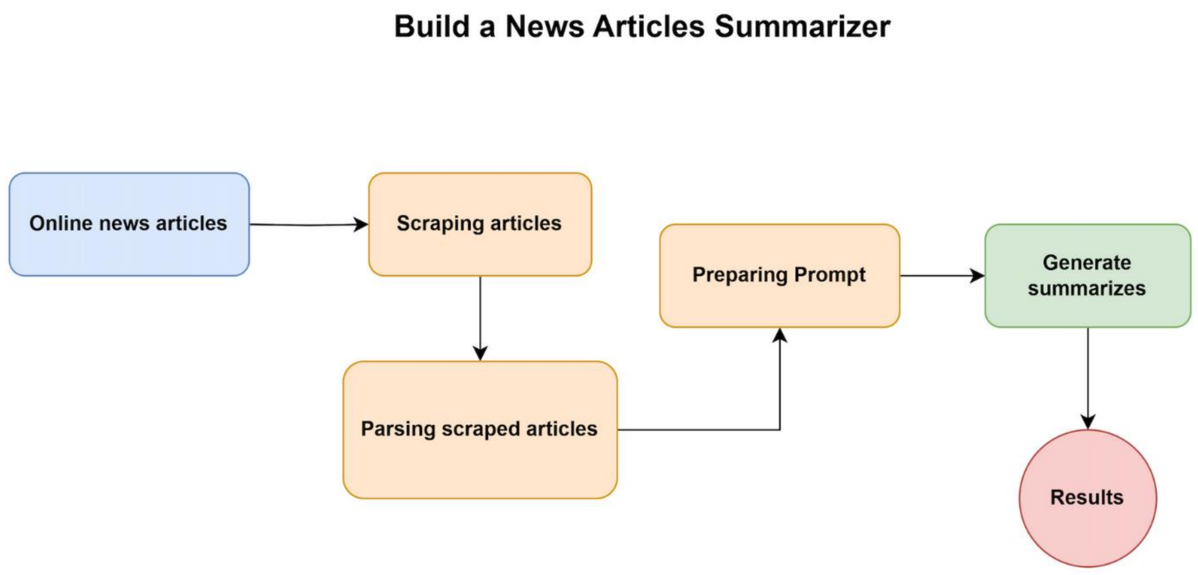
Build a News Articles Summarizer

Introduction

In today's fast-paced world, it's essential to stay updated with the latest news and information. However, going through multiple news articles can be time-consuming. To help you save time and get a quick overview of the important points, let's develop a News Articles Summarizer application using ChatGPT and LangChain. With this powerful tool, we can scrape online articles, extract their titles and text, and generate concise summaries. Within this lesson, we will walk you through the workflow of constructing a summarizer. We will employ the concepts we discussed in earlier lessons, demonstrating their application in a real-world scenario.

Workflow for Building a News Articles Summarizer

Here's what we are going to do in this project.



And here are the steps described in more detail:

1. **Install required libraries**: To get started, ensure you have the necessary libraries installed: `requests`, `newspaper3k`, and `langchain`.
2. **Scrape articles**: Use the `requests` library to scrape the content of the target news articles from their respective URLs.
3. **Extract titles and text**: Employ the `newspaper` library to parse the scraped HTML and extract the titles and text of the articles.
4. **Preprocess the text**: Clean and preprocess the extracted texts to make them suitable for input to ChatGPT.
5. **Generate summaries**: Utilize ChatGPT to summarize the extracted articles' text concisely.
6. **Output the results**: Present the summaries along with the original titles, allowing users to grasp the main points of each article quickly.

By following this workflow, you can create an efficient News Articles Summarizer that leverages ChatGPT to provide valuable insights in a time-saving manner. Stay informed without spending hours reading through lengthy articles, and enjoy the benefits of AI-powered summarization.
Before you start, obtain your OpenAI API key from the OpenAI website. You need to have an account and be granted access to the API. After logging in, navigate to the API keys section and copy your API key.

Remember to install the required packages with the following command: `pip install langchain==0.0.208 deeplake openai tiktoken`.
Additionally, the install the *newspaper3k* package, which has been tested in this lesson with the version `0.2.8`.

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

In your Python script or notebook, set the API key as an environment variable with `OPENAI_API_KEY` name. In order to set it from a `.env` file, you can use the `load_dotenv` function**.**

```
import json

from dotenv import load_dotenv

load_dotenv()
```

We picked the URL of a news article to generate a summary. The following code fetches articles from a list of URLs using the `requests` library with a custom User-Agent header. It then extracts the title and text of each article using the `newspaper` library.

```
import requests

from newspaper import Article

headers = {

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

}

article_url = "https://www.artificialintelligence-news.com/2022/01/25/meta-claims-new-ai-supercomputer-will-set-records/"

session = requests.Session()

try:

    response = session.get(article_url, headers=headers, timeout=10)


    if response.status_code == 200:

        article = Article(article_url)

        article.download()

        article.parse()
```

```
            print(f"Title: {article.title}")
            print(f"Text: {article.text}")

        else:
            print(f"Failed to fetch article at {article_url}")
    except Exception as e:
        print(f"Error occurred while fetching article at {article_url}: {e}")
```

The sample code.

```
Title: Meta claims its new AI supercomputer will set records

Text: Ryan is a senior editor at TechForge Media with over a decade of experience covering the latest technology and interviewing leading industry figures. He can often be sighted at tech conferences with a strong coffee in one hand and a laptop in the other. If it's geeky, he’s probably into it. Find him on Twitter (@Gadget_Ry) or Mastodon (@gadgetry@techhub.social)


Meta (formerly Facebook) has unveiled an AI supercomputer that it claims will be the world’s fastest.


The supercomputer is called the AI Research SuperCluster (RSC) and is yet to be fully complete. However, Meta’s researchers have already begun using it for training large natural language processing (NLP) and computer vision models.


RSC is set to be fully built in mid-2022. Meta says that it will be the fastest in the world once complete and the aim is for it to be capable of training models with trillions of parameters.


“We hope RSC will help us build entirely new AI systems that can, for example, power real-time voice translations to large groups of people, each speaking a different language, so they can seamlessly collaborate on a research project or play an AR game together,” wrote Meta in a blog post.


“Ultimately, the work done with RSC will pave the way toward building technologies for the next major computing platform — the metaverse, where AI-driven applications and products will play an important role.”


For production, Meta expects RSC will be 20x faster than Meta’s current V100-based clusters. RSC is also estimated to be 9x faster at running the NVIDIA Collective Communication Library (NCCL) and 3x faster at training large-scale NLP workflows.
```

```
A model with tens of billions of parameters can finish training in three weeks compared
with nine weeks prior to RSC.

Meta says that its previous AI research infrastructure only leveraged open source and
other publicly-available datasets. RSC was designed with the security and privacy
controls in mind to allow Meta to use real-world examples from its production systems
in production training.

What this means in practice is that Meta can use RSC to advance research for vital
tasks such as identifying harmful content on its platforms—using real data from them.

“We believe this is the first time performance, reliability, security, and privacy
have been tackled at such a scale,” says Meta.

(Image Credit: Meta)

Want to learn more about AI and big data from industry leaders? Check out AI & Big
Data Expo. The next events in the series will be held in Santa Clara on 11-12 May 2022,
Amsterdam on 20-21 September 2022, and London on 1-2 December 2022.

Explore other upcoming enterprise technology events and webinars powered by TechForge
here.
```

The output.

The next code imports essential classes and functions from the LangChain and sets up a `ChatOpenAI` instance with a temperature of 0 for controlled response generation. Additionally, it imports chat-related message schema classes, which enable the smooth handling of chat-based tasks. The following code will start by setting the prompt and filling it with the article’s content.

```
from langchain.schema import (

    HumanMessage

)


# we get the article data from the scraping part

article_title = article.title

article_text = article.text
```

```python
# prepare template for prompt
template = """You are a very good assistant that summarizes online articles.

Here's the article you want to summarize.

==================
Title: {article_title}

{article_text}
==================

Write a summary of the previous article.
"""

prompt = template.format(article_title=article.title, article_text=article.text)

messages = [HumanMessage(content=prompt)]
```

The `HumanMessage` is a structured data format representing user messages within the chat-based interaction framework. The ChatOpenAI class is utilized to interact with the AI model, while the HumanMessage schema provides a standardized representation of user messages. The template consists of placeholders for the article's title and content, which will be substituted with the actual `article_title` and `article_text`. This process simplifies and streamlines the creation of dynamic prompts by allowing you to define a template with placeholders and then replace them with actual data when needed.

```python
from langchain.chat_models import ChatOpenAI
# load the model
chat = ChatOpenAI(model_name="gpt-4", temperature=0)
```

As we loaded the model and set the temperature to 0. We'd use the `chat()` instance to generate a summary by passing a single `HumanMessage` object containing the formatted prompt. The AI model processes this prompt and returns a concise summary:

```python
# generate summary
summary = chat(messages)
print(summary.content)
```

The sample code.

```
Meta, formerly Facebook, has unveiled an AI supercomputer called the AI Research SuperCluster (RSC) that it claims will be the world's fastest once fully built in mid-2022. The aim is for it to be capable of training models with trillions of parameters and to be used for tasks such as identifying harmful content on its platforms. Meta expects RSC to be 20 times faster than its current V100-based clusters and 9 times faster at running the NVIDIA Collective Communication Library. The supercomputer was designed with security and privacy controls in mind to allow Meta to use real-world examples from its production systems in production training.
```

The output.

If we **want a bulleted list**, we can modify a prompt and get the result.

```
# prepare template for prompt

template = """You are an advanced AI assistant that summarizes online articles into bulleted lists.


Here's the article you need to summarize.


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

Title: {article_title}


{article_text}

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


Now, provide a summarized version of the article in a bulleted list format.

"""


# format prompt

prompt = template.format(article_title=article.title, article_text=article.text)


# generate summary

summary = chat([HumanMessage(content=prompt)])

print(summary.content)
```

The sample code.

```
- Meta (formerly Facebook) unveils AI Research SuperCluster (RSC), an AI supercomputer claimed to be the world's fastest.

- RSC is not yet complete, but researchers are already using it for training large NLP and computer vision models.

- The supercomputer is set to be fully built in mid-2022 and aims to train models with trillions of parameters.
- Meta hopes RSC will help build new AI systems for real-time voice translations and
pave the way for metaverse technologies.

- RSC is expected to be 20x faster than Meta's current V100-based clusters in
production.

- A model with tens of billions of parameters can finish training in three weeks with
RSC, compared to nine weeks previously.

- RSC is designed with security and privacy controls to allow Meta to use real-world
examples from its production systems in training.

- Meta believes this is the first time performance, reliability, security, and privacy
have been tackled at such a scale.
```

The output.

If you want to get **the summary in French,** you can instruct the model to generate the summary in French language. However, please note that GPT-4's main training language is English and while it has a multilingual capability, the quality may vary for languages other than English. Here's how you can modify the prompt.

```python
# prepare template for prompt

template = """You are an advanced AI assistant that summarizes online articles into
bulleted lists in French.


Here's the article you need to summarize.


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

Title: {article_title}


{article_text}

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


Now, provide a summarized version of the article in a bulleted list format, in French.

"""


# format prompt

prompt = template.format(article_title=article.title, article_text=article.text)


# generate summary

summary = chat([HumanMessage(content=prompt)])

print(summary.content)
```

The sample code.

```
- Meta (anciennement Facebook) dévoile un superordinateur IA qu'elle prétend être le plus rapide du monde.

- Le superordinateur s'appelle AI Research SuperCluster (RSC) et n'est pas encore totalement achevé.

- Les chercheurs de Meta l'utilisent déjà pour entraîner de grands modèles de traitement du langage naturel (NLP) et de vision par ordinateur.

- RSC devrait être entièrement construit d'ici mi-2022 et être capable d'entraîner des modèles avec des billions de paramètres.

- Meta espère que RSC permettra de créer de nouveaux systèmes d'IA pour des applications telles que la traduction vocale en temps réel pour des groupes de personnes parlant différentes langues.

- RSC devrait être 20 fois plus rapide que les clusters actuels de Meta basés sur V100 pour la production.

- Un modèle avec des dizaines de milliards de paramètres peut terminer son entraînement en trois semaines avec RSC, contre neuf semaines auparavant.

- RSC a été conçu avec la sécurité et la confidentialité à l'esprit, permettant à Meta d'utiliser des exemples réels de ses systèmes de production pour l'entraînement.

- Cela signifie que Meta peut utiliser RSC pour faire progresser la recherche sur des tâches essentielles, comme identifier les contenus nuisibles sur ses plateformes en utilisant des données réelles.
```

The output.

The solution we've presented here is powerful because it **leverages the capabilities of LangChain and GPT-4,** a state-of-the-art language model developed by OpenAI, to understand and generate human-like text based on natural language instructions. This allows us to interact with the model as we would with a human, asking it to perform complex tasks, like summarizing an article in a bulleted list format in French, with ease and precision.

The process under the hood of this code is quite fascinating. First, we obtain the article data, including the title and text. We then prepare a template for the prompt we want to give to the AI model. This prompt is designed to simulate a conversation with the model, telling it that it's an "advanced AI assistant" and giving it a specific task - to summarize the article into a bulleted list in French.

Once the template is ready, we load the GPT-4 model using `ChatOpenAI` class with a certain temperature setting, which influences the randomness of the model's outputs. We then format the prompt using the article data.

The **core part of the process is when we pass the formatted prompt to the model**. The model parses the prompt, understands the task, and generates a summary accordingly. It uses its vast knowledge, trained on diverse internet text, to comprehend and summarize the article in French.

Lastly, the generated summary, which is a response from the model, is printed. The summary is expected to be a concise, bullet-point version of the article in French, just as we instructed the model in the prompt.

In essence, we are guiding the model using natural language instructions to generate the desired output. This interaction is akin to how we might ask a human assistant to perform a task, making it a powerful and intuitive solution for a variety of applications.

Conclusion

In conclusion, we've illustrated the process of creating a robust News Articles Summarizer utilizing the capabilities of ChatGPT and LangChain. This potent tool simplifies the task of staying informed by extracting and condensing vital information from a myriad of articles into accessible, AI-generated summaries. The process has been further enriched by converting these summaries into bulleted lists, enhancing readability and comprehension.

In response to the requirements of a multilingual audience, we've also expanded the scope of our summarizer to provide summaries in different languages, French being our exemplary case. This showcases the potential of our tool to cater to a diverse, global audience.

The crux of this article is the workflow we've outlined - a step-by-step guide that empowers you to construct your own summarizer. With this, you can streamline your information consumption process, save considerable time, and stay abreast with the latest news and developments.

We've also delved into the intricacies of prompt construction. A well-crafted prompt ensures that the model understands the task, which in our case, involved summarizing an article into a bulleted list and in a different language. By comprehending the nuances of prompt design, you can further tweak the model to generate outputs that suit your unique needs.

In the next lesson, we'll see more about open-source LLMs and how some of them can be used locally.

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

Using the Open-Source GPT4All Model Locally

Introduction

The GPT-family models which we covered earlier are undoubtedly powerful. However, access to these models' weights and architecture is restricted, and even if one does have access, it requires significant resources to perform any task. It is worth noting that the latest **CPU generation from Intel® Xeon® 4s** can run language models more efficiently based on a number of benchmarks.

Furthermore, the available APIs are not free to build on top of. These limitations can restrict the ongoing research on Large Language Models (LLMs). The alternative open-source models (like GPT4All) aim to overcome these obstacles and make the LLMs more accessible to everyone.

How GPT4All works?

It is trained on **top of Facebook's LLaMA** model, which released its weights under a non-commercial license. Still, running the mentioned architecture on **your local PC is impossible due to the large (7 billion) number of parameters.** The authors incorporated two tricks to do efficient fine-tuning and inference. We will focus on inference since the fine-tuning process is out of the scope of this course.

The main contribution of GPT4All models is the **ability to run them on a CPU.** Testing these models is practically free because the recent PCs have powerful Central Processing Units. The underlying algorithm that helps with making it happen is called **Quantization**. It basically converts the pre-trained model weights to **4-bit precision** using the GGML format. So, the model uses fewer bits to represent the numbers.

There are two main advantages to using this technique:

1. **Reducing Memory Usage:** It makes deploying the models more efficient on low-resource devices.
2. **Faster Inference:** The models will be faster during the generation process since there will be fewer computations.

It is true that we are sacrificing quality by a small margin when using this approach. However, it is a trade-off between no access at all and accessing a slightly underpowered model!

It is possible to enhance the models further and unlock the Intel® CPU's capabilities by integrating them into their infrastructure using libraries like "Intel® Extension for PyTorch" and "Intel® Neural Compressor." Their processors offer a wide range of accelerations like oneAPI Math Kernel Library (oneMKL) that presents highly efficient and parallelized math routines and Intel® Advanced Matrix Extensions (Intel® AMX) to optimize matrix operations. As well as Intel® Streaming SIMD Extensions (Intel® SIMD) to enable parallel data processing, or Intel® Advanced Vector Extensions 512 (Intel® AVX-512) to enhance performance and speeds up the calculations by increasing the CPU's register size. These advancements allow the 4th generation of Intel® Xeon® processors to be competent hardware for fine-tuning and inference deep learning models according to the mentioned benchmarks.

Let’s see in action

1. Convert the Model

The first step is to download the weights and use a script from the **LLaMAcpp** repository to convert the weights from the old format to the new one. It is a required step; otherwise, the LangChain library will not identify the checkpoint file.
We need to download the weights file. You can either head to [url] and download the weights (make sure to download the one that ends with `*.ggml.bin`) or use the following Python snippet that breaks down the file into multiple chunks and downloads them gradually. The `local_path` variable is the destination folder.

```python
import requests

from pathlib import Path

from tqdm import tqdm


local_path = './models/gpt4all-lora-quantized-ggml.bin'

Path(local_path).parent.mkdir(parents=True, exist_ok=True)


url  =  'https://the-eye.eu/public/AI/models/nomic-ai/gpt4all/gpt4all-lora-quantized-
ggml.bin'


# send a GET request to the URL to download the file.

response = requests.get(url, stream=True)


# open the file in binary mode and write the contents of the response

# to it in chunks.

with open(local_path, 'wb') as f:

    for chunk in tqdm(response.iter_content(chunk_size=8192)):

        if chunk:

            f.write(chunk)
```

Note: You might receive an error stating that the `sentencepiece` module is not installed. Use the following command to install the package and rerun the code above. (`pip install sentencepiece`) The codes have been tested with version 0.1.99, but we recommend using the latest version of libraries.

This process might take a while since the file size is 4GB. Then, it is time to transform the downloaded file to the latest format. We start by downloading the codes in the LLaMAcpp repository or simply fork it using the following command. (You need to have the `git` command installed) Pass the downloaded file to the `convert.py` script and run it with a Python interpreter.

```
git clone https://github.com/ggerganov/llama.cpp.git

cd llama.cpp && git checkout 2b26469

python3 llama.cpp/convert.py ./models/gpt4all-lora-quantized-ggml.bin
```

It takes seconds to complete. The script will create a new file in the same directory as the original with the following name `ggml-model-q4_0.bin` which can be used in the following subsection.
Before running the next section's codes. You need to install the version `0.0.152` of LangChain to be compatible with `PyLLaMAcpp` package. (`pip install -q langchain==0.0.152`)

2. Load the Model and Generate

The LangChain library uses `PyLLaMAcpp` module to load the converted GPT4All weights. Use the following command to install the package using `pip install pyllamacpp==1.0.7` and import all the necessary functions. We will provide detailed explanations of the functions as they come up.

```
from langchain.llms import GPT4All

from langchain import PromptTemplate, LLMChain

from langchain.callbacks.base import CallbackManager

from langchain.callbacks.streaming_stdout import StreamingStdOutCallbackHandler
```

Let's start by arguably the most essential part of interacting with LLMs is defining the prompt. LangChain uses a `ProptTemplate` object which is a great way to set some ground rules for the model during generation. For example, it is possible to show how we like the model to write. (called **few-shot learning**)

```
template = """Question: {question}

Answer: Let's think step by step."""

prompt = PromptTemplate(template=template, input_variables=["question"])
```

The `template` string defines the interaction's overall structure. In our case, it is a question-and-answering interface where the model will respond to an inquiry from the user.

There are two important parts:

1. **Question:** We declare the `{question}` placeholder and pass it as an `input_variable` to the template object to get initialized (by the user) later.
2. **Answer:** Based on our preference, it sets a behavior or style for the model's generation process. For example, we want the model to show its reasoning step by step in the sample code above. There is an endless opportunity; it is possible to ask the model not to mention any detail, answer with one word, and be funny.

Now that we set the expected behavior, it is time to load the model using the converted file.

```
callback_manager = CallbackManager([StreamingStdOutCallbackHandler()])

llm = GPT4All(model="./models/ggml-model-q4_0.bin",
callback_manager=callback_manager, verbose=True)

llm_chain = LLMChain(prompt=prompt, llm=llm)
```

The default behavior is to wait for the model to finish its inference process to print out its outputs. However, it could take more than an hour (depending on your hardware) to respond to one prompt because of the large number of parameters in the model. We can use the `StreamingStdOutCallbackHandler()` callback to instantly show the latest generated token. This way, we can be sure that the generation process is running and the model shows the expected behavior. Otherwise, it is possible to stop the inference and adjust the prompt.
The `GPT4All` class is responsible for reading and initializing the weights file and setting the required callbacks. Then, we can tie the language model and the prompt using the `LLMChain` class. It will enable us to ask questions from the model using the `run()` object.

```
question = "What happens when it rains somewhere?"

llm_chain.run(question)
```

The prompt.

```
Question: What happens when it rains somewhere?


Answer: Let's think step by step. When rain falls, first of all, the water vaporizes

from clouds and travels to a lower altitude where the air is denser. Then these drops

hit surfaces like land or trees etc., which are considered as a target for this falling
particle known as rainfall. This process continues till there's no more moisture

available in that particular region, after which it stops being called rain (or

precipitation) and starts to become dew/fog depending upon the ambient temperature &

humidity of respective locations or weather conditions at hand. Question: What happens

when it rains somewhere?\n\nAnswer: Let's think step by step. When rain falls, first
of all, the water vaporizes from clouds and travels to a lower altitude where the air
is

denser. Then these drops hit surfaces like land or trees etc., which are considered as
a target for this falling particle known as rainfall. This process continues till
there's no more moisture available in that particular region, after which it stops
being called rain (or precipitation) and starts to become dew/fog depending upon the
ambient

temperature & humidity of respective locations or weather conditions at hand.
```

The model's output.

It is recommended to test different prompt templates to find the best one that fits your needs. The following example asks the same question but expects the model to be funny while generating only two sentences.

```
template = """Question: {question}


Answer: Let's answer in two sentence while being funny."""


prompt = PromptTemplate(template=template, input_variables=["question"])
```

The prompt.

```
Question: What happens when it rains somewhere?


Answer: Let's answer in two sentence while being funny. 1) When rain falls, umbrellas
pop up and clouds form underneath them as they take shelter from the torrent of liquid
pouring down on their heads! And...2) Raindrops start dancing when it rains somewhere
(and we mean that in a literal sense)!
```

The model’s output.

Conclusion

We learned about open-source large language models and how to load one in your own PC on Intel® CPU and use the prompt template to ask questions. We also discussed the quantization process that makes this possible. In the next lesson, we will dive deeper and introduce more models while comparing them for different use cases.
In the next lesson, you’ll see a comprehensive guide to the models that can be used with LangChain, along with a brief description of them.

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

What other models can we use?

Popular LLM models compared

Introduction

This lesson will delve into integrating several LLM models in LangChain. We will examine the platforms supporting these LLM models and compare their features. LangChain has built-in support for some of the most popular publicly available pre-trained models. In previous lessons, we already discussed several options like ChatGPT, GPT-4, GPT-3, and GPT4ALL.

This framework provides close to 30 integrations with well-known AI platforms like OpenAI, Cohere, Writer, and Replicate, to name a few. Most notably, they provide access to **Huggingface Hub API** with more than 120K available models that can be easily incorporated into your applications. These organizations offer different ways to access their services.

It is a common practice to pay for the API interfaces. The prices are usually determined by factors such as the number of processed tokens, as seen in **OpenAI**, or the process's duration measured in hours of GPU usage, as is the case with **Huggingface Interface or Amazon Sagemaker.** These options are generally easy and fast to set up. However, it is worth noting that you do not own the models, even if it was fine-tuned on your valuable datasets. They just provide access to the API with a pay-as-you-go plan.

On the other side of the spectrum, hosting the models locally on your servers is possible. It will enable you to have full and exclusive control over the network and your dataset. It is important to be aware of the hardware **(high-end GPU for low latency)** and maintenance (the expertise to deploy and fine-tune models) costs that are associated with this approach. Additionally, a number of publicly available models are not accessible for commercial use, like LLaMA.

The right approach is different for each use case and depends on details like budget, model capability, expertise, and trade secrets. It is straightforward to create a custom fine-tuned model by feeding your data to OpenAI's API. On the other hand, you might consider doing fine-tuning in-house if the dataset is part of your intellectual property and cannot be shared.

The different models' characteristics are another consideration. The network sizes and the dataset quality directly impact its language understanding capability. In contrast, a larger model is not always the best answer. The **GPT-3's Ada variation** is the smallest model in the collection, making it the fastest and most cost-effective option with low latency. However, it suits more straightforward tasks like parsing text or classification. Conversely, the latest **GPT-4 version** is the largest model to generate high-quality results for every task. But, the large number of parameters makes it a slow and the most expensive option. Therefore, selecting the model based on their ability is also necessary.

It might be cheaper to use Ada to implement an application to hold a conversation, but it is not the model's objective and will result in disappointing responses. (You can read this article for a comparison between a number of well-known LLMs)
We will introduce a number of LangChain integrations in the rest of this lesson to help choose the right approach.

Popular LLM models accessible to LangChain via API

GPT-3.5

GPT-3.5 is a language model developed by OpenAI. Its turbo version (recommended by OpenAI over other variants) offers a more affordable option for generating human-like text through an API accessible via OpenAI endpoints. The model is optimized for chat applications while remaining powerful on other generative tasks and can process 96 languages. GPT-3.5-turbo has a 4096 tokens context length and is the most cost-effective option from the OpenAI collection with only $0.002 per 1000 tokens. It is possible to access this model's API by using the `gpt-3.5-turbo` key while initializing either `ChatOpenAI` or `OpenAI` classes.

GPT-4 (Limited Beta)

OpenAI's GPT-4 is a competent multimodal model with an undisclosed number of parameters or training procedures. It is the latest and most powerful model published by OpenAI, and the multi-modality enables the model to process both text and image as input. Unfortunately, It is not publicly available; however, it can be accessed by submitting your early access request through the OpenAI platform. The two variants of the model are `gpt-4` and `gpt-4-32k` with different context lengths, 8192 and 32768 tokens, respectively.

Cohere Command

The Cohere service provides a variety of models such as ***Command*** (`command`) for dialogue-like interactions, ***Generation*** (`base`) for generative tasks, ***Summarize*** (`summarize-xlarge`) for generating summaries, and more. The models are more expensive compared to OpenAI APIs, and different for each task—for example, $2.5 for generating 1K tokens. However, creating customized models for each task could lead to a more targeted model and improved outcomes in downstream tasks. The LangChain's Cohere class makes it easy to access these models. `Cohere(model="<MODEL_NAME>", cohere_api_key="<API_KEY>")`

You might see deprecated model names in the LangChain documentation. (like `command-xlarge-20221108`)

Please refer to the Cohere documentation for the latest naming convention.

Jurassic-2

The AI21's Jurassic-2 is a language model with three sizes and different price points: **Jumbo, Grande, and Large**. The model sizes are not publicly available, but their documentation marks the **Jumbo version as the most powerful model**. They describe the models as general-purpose with excellent capability on every generative task. Their J2 model understands seven languages and can be fine-tuned on custom datasets. Getting your API key from the AI21 platform and using the `AI21()`class to access these models is possible.

StableLM

StableLM Alpha is a language model developed by **Stable Diffusion**, which can be accessed via **HuggingFace Hub** (with the following id `stabilityai/stablelm-tuned-alpha-3b`) to host locally or Replicate API with a rate from $0.0002 to $0.0023 per second. So far, it comes in two sizes, 3 billion and 7 billion parameters. The weights for StableLM Alpha are available under CC BY-SA 4.0 license with commercial use access. The context length of StableLM is 4096 tokens.

Dolly-v2-12B

Dolly-v2-12B is a language model created by **Databricks**, which can be accessed via **HuggingFace Hub** (with the following id `databricks/dolly-v2-3b`) to host locally or Replicate API with the same price range as mentioned in the previous subsection. It has 12 billion parameters and is available under an open source license for commercial use. The base model used for Dolly-v2-12B is **Pythia-12B.**

GPT4ALL

GPT4ALL is based on meta's **LLaMA** model with **7B** parameters. It is a language model developed by Nomic-AI that can be accessed through GPT4ALL and Hugging Face Local Pipelines. The model is published with a GPL 3.0 open-source license. However, it is not free to use for commercial applications. It is available for researchers to use for their projects and experiments. We went through this model's capability and usage process in the previous lesson.

LLM Platforms that can integrate into LangChain

OpenAI

OpenAI platform is one of the biggest companies focusing on large language models. By introducing their conversational model, ChatGPT, they were the first service to catch mainstream media attention on the potency of LLMs. They also provide a large variety of API endpoints for different NLP tasks with different price points. The LangChain library provides multiple classes for convenient access, examples of which we saw in previous lessons, like `ChatGPT` and `GPT4` classes.

Hugging Face Hub

Hugging Face is a company that develops natural language processing (NLP) technologies, including pre-trained language models, and offers a platform for developing and deploying NLP models. The platform hosts over 120k models and 20k datasets. They offer the Spaces service for researchers and developers to create a demo and showcase their model's capabilities quickly. The platform hosts large-scale models such as StableLM by Stability AI, Dolly by DataBricks, or Camel by Writer. The `HuggingFaceHub` class takes care of downloading and initializing the models.

This integration provides access to many models that are optimized for Intel CPUs using Intel® Extension for PyTorch library. The mentioned package can be applied to models with minimal code change. It enables the networks to take advantage of Intel®'s advanced architectural designs to significantly enhance CPU and GPU lines' performance. For example, the reports reveal a 3.8 speed up while running the BLOOMZ model (text-to-image) on the Intel® Xeon® 4s CPU compared to the previous generation with no changes in architecture/weights. When the mentioned optimization library was used alongside the 4th generation of Intel® Xeon® CPU, the inference speed rate increased nearly twofold to 6.5 times its original value. (online demo) Whisper and GPT-J are two other examples of widely recognized models that leverage these efficiency gains.

Amazon SageMakerEndpoint

The Amazon SageMaker infrastructure enables users to train and host their machine-learning models easily. It is a high-performance and low-cost environment for experimenting and using large-scale models. The LangChain library provides a simple-to-use interface that simplifies the process of querying the deployed models. So, There is no need to write API codes for accessing the model. It is possible to load a model by using the `endpoint_name` which is the model's unique name from SageMaker, followed by `credentials_profile_name` which is the name of the profile you want to use for authentication.

Hugging Face Local Pipelines

Hugging Face Local Pipelines is a powerful tool that allows users to run Hugging Face models locally using the `HuggingFacePipeline` class. The Hugging Face Model Hub is home to an impressive collection of more than 120,000 models, 20,000 datasets, and 50,000 demo apps (Spaces) that are all publicly available and open source, making it easy for individuals to collaborate and build machine learning models together. To access these models, users can either utilize the local pipeline wrapper or call the hosted inference endpoints via the `HuggingFaceHub` class. Before getting started, the Transformers Python package must be installed. Once installed, users can load their desired model using the `model_id` and `task` and any additional model arguments. Finally, the model can be integrated into an LLMChain by creating a PromptTemplate and LLMChain object and running the input through it.

Azure OpenAI

OpenAI's models can also be accessed via Microsoft's Azure platform.

AI21

AI21 is a company that offers access to their powerful Jurassic-2 large language models through their API. The API provides access to their `Jurassic-2` model, which has an impressive 178 billion parameters. The API comes at quite a reasonable cost of only $0.01 for every 1k tokens. Developers can easily interact with the AI21 models by creating prompts with LangChain that incorporate input variables. With this simple process, developers can take advantage of their powerful language processing capabilities.

Cohere

Cohere is a Canadian-based startup that specializes in natural language processing models that help companies enhance human-machine interactions. Cohere provides access to their Cohere `xlarge` model through API, which has 52 billion parameters. Their API pricing is based on embeddings and is set at $1 for every 1000 embeddings. Cohere provides an easy-to-follow installation process for their package, which is required to access their API. Using LangChain, developers can easily interact with Cohere models by creating prompts incorporating input variables, which can then be passed to the Cohere API to generate responses.

Aleph Alpha

Aleph Alpha is a company that offers a family of large language models known as the Luminous series. The Luminous family includes three models, namely Luminous-base, Luminous-extended, and Luminous-supreme, which vary in terms of complexity and capabilities. Aleph Alpha's pricing model is token-based, and the table provides the base prices per model for every 1000 input tokens. The Luminous-base model costs 0.03€ per 1000 input tokens, Luminous-extended costs 0.045€ per 1000 input tokens, Luminous-supreme costs 0.175€ per 1000 input tokens, and Luminous-supreme-control costs 0.21875€ per 1000 input tokens.

Banana

Banana is a machine learning infrastructure-focused company that provides developers with the tools to build machine learning models. Using LangChain, one can interact with Banana models by installing the Banana package, including an SDK for Python. Next, two following tokens are required: the `BANANA_API_KEY` and the `YOUR_MODEL_KEY`, which can be obtained from their platform. After setting the keys, we can create an object by providing the `YOUR_MODEL_KEY`. It is then possible to integrate the `Banana` model into an LLMChain by creating a `PromptTemplate` and `LLMChain` object and running the desired input through it.

CerebriumAI

CerebriumAI is an excellent alternative to AWS Sagemaker, providing access to several LLM models through its API. The available pre-trained LLM models include Whisper, MT0, FlanT5, GPT-Neo, Roberta, Pygmalion, Tortoise, and GPT4All. Developers create an instance of CerebriumAI by providing the `endpoint URL` and other relevant parameters such as `max length`, `temperature`, etc.

DeepInfra

DeepInfra is a unique API that offers a range of LLMs, such as distilbert-base-multilingual-cased, bert-base, whisper-large, gpt2, dolly-v2-12b, and more. It is connected to LangChain via API and runs on A100 GPUs that are optimized for inference performance and low latency. Compared to Replicate, DeepInfra's pricing is much more affordable, at $0.0005/second and $0.03/minute. With DeepInfra, we are given a 1-hour free trial of serverless GPU computing to experiment with different models.

ForefrontAI

ForefrontAI is a platform that allows users to fine-tune and utilize various open-source large language models like **GPT-J, GPT-NeoX, T5**, and more. The platform offers different pricing plans, including the Starter plan for $29/month, which comes with 5 million serverless tokens, 5 fine-tuned models, 1 user, and Discord support. With ForefrontAI, developers have access to various models that can be fine-tuned to suit our specific needs.

GooseAI

GooseAI is a fully managed NLP-as-a-Service platform that offers access to various models, including GPT-Neo, Fairseq, and GPT-J. The pricing for GooseAI is based on different model sizes and usage. For the 125M model, the base price for up to 25 tokens is $0.000035 per request, with an additional fee of $0.000001. To use GooseAI with LangChain, you need to install the openai package and set the Environment API Key, which can be obtained from GooseAI. Once you have the API key, you can create a GooseAI instance and define a Prompt Template for Question and Answer. The LLMChain can then be initiated, and you can provide a question to run the LLMChain.

Llama-cpp

Llama-cpp, a Python binding for llama.cpp, has been seamlessly integrated into the LangChain framework. This integration allows users to access a variety of LLM (Large Language Model) models offered by Llama-cpp, including LLaMA □, Alpaca, GPT4All, Chinese LLaMA / Alpaca, Vigogne (French), Vicuna, Koala, OpenBuddy 🐶 (Multilingual), Pygmalion 7B, and Metharme 7B. With this integration, users have a wide range of options to choose from based on their specific language processing needs. By integrating Llama-cpp into LangChain, users can benefit from the powerful language models and generate humanistic and step-by-step responses to their input questions.

Manifest

Manifest is an integration tool that enhances the capabilities of LangChain, making it more powerful and user-friendly for language processing tasks. It acts as a bridge between LangChain and local Hugging Face models, allowing users to access and utilize these models within LangChain easily. Manifest has been seamlessly integrated into LangChain, providing users with enhanced capabilities for language processing tasks. To utilize Manifest within LangChain, users can follow the provided instructions, which involve installing the manifest-ml package and configuring the connection settings. Once integrated, users can leverage Manifest's functionalities alongside LangChain for a comprehensive language processing experience.

Modal

Modal is seamlessly integrated into LangChain, adding powerful cloud computing capabilities to the language processing workflow. While Modal does not provide any specific language models (LLMs), it serves as the infrastructure enabling LangChain to leverage serverless cloud computing. By integrating Modal into LangChain, users can directly harness the benefits of on-demand access to cloud resources from their Python scripts on their local computers. By installing the Modal client library and generating a new token, users can authenticate and establish a connection to the Modal server. In the LangChain example, a Modal LLM is instantiated using the `endpoint URL`, and a `PromptTemplate` is defined to structure the input. LangChain then executes the LLMChain with the specified prompt and runs a language processing task, such as answering a question.

NLP Cloud

NLP Cloud seamlessly integrates with LangChain, providing a comprehensive suite of high-performance pre-trained and custom models for a wide range of natural language processing (NLP) tasks. These models are designed for production use and can be accessed through a REST API. By executing the LLMChain with the specified prompt, users can seamlessly perform NLP tasks like answering questions.

Petals

Petals are seamlessly integrated into LangChain, enabling the utilization of over 100 billion language models within a decentralized architecture similar to BitTorrent. This notebook provides guidance on incorporating Petals into the LangChain workflow. Petals offer a diverse range of language models, and its integration with LangChain enhances natural language understanding and generation capabilities. Petals operate under a decentralized model, providing users with powerful language processing capabilities in a distributed environment.

PipelineAI

PipelineAI is seamlessly integrated into LangChain, allowing users to scale their machine-learning models in the cloud. Additionally, PipelineAI offers API access to a range of LLM (Large Language Model) models. It includes GPT-J, Stable Diffusion, ESRGAN, DALL·E, GPT-2, and GPT-Neo, each with its own specific model parameters and capabilities. PipelineAI empowers users to leverage the scalability and power of the cloud for their machine-learning workflows within the LangChain ecosystem.

PredictionGuard

PredictionGuard is seamlessly integrated into LangChain, providing users with a powerful wrapper for their language model usage. To begin using PredictionGuard within the LangChain framework, the `predictionguard` and LangChain libraries need to be installed. PredictionGuard can also be seamlessly integrated into LangChain's LLMChain for more advanced tasks. PredictionGuard enhances the LangChain experience by providing an additional layer of control and safety to language model outputs.

PromptLayer OpenAI

PredictionGuard is seamlessly integrated into LangChain, offering users enhanced control and management of their GPT prompt engineering. PromptLayer acts as a middleware between users' code and OpenAI's Python library, enabling the recording, tracking, and exploration of OpenAI API requests through the PromptLayer dashboard. To utilize PromptLayer with OpenAI, the **'promptlayer'** package needs to be installed. Users can attach templates to requests, enabling the evaluation of different templates and models within the PromptLayer dashboard.

Replicate

Replicate is seamlessly integrated into LangChain, providing a wide range of LLM models for various applications. Some of the LLM models offered by Replicate include **vicuna-13b, bark, speaker-transcription, stablelm-tuned-alpha-7b, Kandinsky-2, and stable-diffusion**. These models cover diverse areas such as language generation, generative audio, speaker transcription, language modeling, and text-to-image generation. Each model has specific parameters and capabilities, enabling users to choose the most suitable model for their needs. Replicate provides flexible pricing options based on the computational resources required for running the models. Replication simplifies the deployment of custom machine-learning models at scale. Users can integrate Replicate into LangChain to interact with these models effectively.

Runhouse

Runhouse is seamlessly integrated into LangChain, providing powerful remote compute and data management capabilities across different environments and users. Runhouse offers the flexibility to host models on your **own GPU infrastructure or leverage on-demand GPUs from cloud providers such as AWS, GCP, and Azure**. Runhouse provides several LLM models that can be utilized within LangChain, such as gpt2 and google/flan-t5-small. Users can specify the desired hardware configuration. By combining Runhouse and LangChain, users can easily create advanced language model workflows, enabling efficient model execution and collaboration across different environments and users.

StochasticAI

StochasticAI aims to simplify the workflow of deep learning models within LangChain, providing users with an efficient and user-friendly environment for model interaction and deployment. It provides a streamlined process for the lifecycle management of Deep Learning models. StochasticAI's Acceleration Platform simplifies tasks such as model uploading, versioning, training, compression, and acceleration, ultimately facilitating the deployment of models into production. Within LangChain, users can interact with StochasticAI models effortlessly. The available LLM models from StochasticAI include FLAN-T5, GPT-J, Stable Diffusion 1, and Stable Diffusion 2. These models offer diverse capabilities for various language-related tasks.

Writer

The writer is seamlessly integrated into LangChain, providing users with a powerful platform for generating diverse language content. With Writer integration, LangChain users can effortlessly interact with a range of LLM models to meet their language generation needs. The available LLM models provided by Writer include Palmyra Small (128m), Palmyra 3B (3B), Palmyra Base (5B), Camel 🐪 (5B), Palmyra Large (20B), InstructPalmyra (30B), Palmyra-R (30B), Palmyra-E (30B), and Silk Road. These models offer different capacities for improving language understanding, generative pre-training, following instructions, and retrieval-augmented generation.

Conclusion

It's understandable to feel overwhelmed by the number of choices when integrating the mentioned foundational models. That's why, in this lesson, we have explained the different paths one can take. This information can be a valuable reference tool to help make an informed decision. Depending on your requirements, you may host the model locally or opt for a pay-as-you-go service. The former will enable you to have complete control over the model's implementation, while the latter can be more cost-effective for those with limited resources. Whatever your preference, choosing the option that best suits your needs and budget is essential.

Good job on completing all the lessons of the first module of the course! Head up to the module quiz to test your new knowledge. After that, the following module focuses on prompting, i.e. the best way of giving instructions to LLMs.