LangChain: Introduction and Getting Started

Large **L**anguage **M**odels (LLMs) entered the world stage with the release of OpenAI's GPT-3 in 2020 [1]. Since then, they've enjoyed a steady growth in popularity.

That is until late 2022. Interest in LLMs and the broader discipline of generative AI has skyrocketed. The reasons for this are likely the continuous upward momentum of significant advances in LLMs.

We saw the dramatic news about Google's *"sentient"* LaMDA chatbot. The first high-performance and *open-source* LLM called BLOOM was released. OpenAI released their next-generation text embedding model and the next generation of *"GPT-3.5"* models.

After all these giant leaps forward in the LLM space, OpenAI released *ChatGPT* — thrusting LLMs into the spotlight.

LangChain appeared around the same time. Its creator, Harrison Chase, made the first commit in late October 2022. Leaving a short couple of months of development before getting caught in the LLM wave.

Despite being early days for the library, it is already packed full of incredible features for building amazing tools around the core of LLMs. In this article, we'll introduce the library and start with the most straightforward component offered by LangChain — LLMs.

LangChain

At its core, LangChain is a framework built around LLMs. We can use it for chatbots, **G**enerative **Q**uestion-**A**nswering (GQA), summarization, and much more.

The core idea of the library is that we can *"chain"* together different components to create more advanced use cases around LLMs. Chains may consist of multiple components from several modules:

- **Prompt templates**: Prompt templates are templates for different types of prompts. Like "chatbot" style templates, ELI5 question-answering, etc

- **LLMs**: Large language models like GPT-3, BLOOM, etc

- **Agents**: Agents use LLMs to decide what actions should be taken. Tools like web search or calculators can be used, and all are packaged into a logical loop of operations.

- **Memory**: Short-term memory, long-term memory.

We will dive into each of these in much more detail in upcoming chapters of the LangChain handbook. You can stay updated for each release via our newsletter:

Subscribe to stay updated with LangChain releases!Submit

For now, we'll start with the basics behind **prompt templates** and **LLMs**. We'll also explore two LLM options available from the library, using models from *Hugging Face Hub* or *OpenAI*.

Our First Prompt Templates

Prompts being input to LLMs are often structured in different ways so that we can get different results. For Q&A, we could take a user's question and reformat it for different Q&A styles, like conventional Q&A, a bullet list of answers, or even a summary of problems relevant to the given question.

Creating Prompts in LangChain

Let's put together a simple question-answering prompt template. We first need to install the `langchain` library.

```
!pip install langchain
```

Follow along with the code via Colab!

From here, we import the `PromptTemplate` class and initialize a template like so:

```
from langchain import PromptTemplate

template = """Question: {question} Answer: """
prompt = PromptTemplate(
template=template,
input_variables=['question']
)

# user question
question = "Which NFL team won the Super Bowl in the 2010 season?"
```

Copy

When using these prompt template with the given `question` we will get:

```
Question: Which NFL team won the Super Bowl in the 2010 season? Answer:
```

For now, that's all we need. We'll use the same prompt template across both Hugging Face Hub and OpenAI LLM generations.

Hugging Face Hub LLM

The Hugging Face Hub endpoint in LangChain connects to the Hugging Face Hub and runs the models via their free inference endpoints. We need a Hugging Face account and API key to use these endpoints.

Once you have an API key, we add it to the `HUGGINGFACEHUB_API_TOKEN` environment variable. We can do this with Python like so:

```
import os

os.environ['HUGGINGFACEHUB_API_TOKEN'] = 'HF_API_KEY'
```

Copy

Next, we must install the `huggingface_hub` library via Pip.

```
!pip install huggingface_hub
```

Now we can generate text using a Hub model. We'll use `google/flan-t5-xl` .

The default Hugging Face Hub inference APIs do not use specialized hardware and, therefore, can be slow. They are also not suitable for running larger models like `bigscience/bloom-560m` or `google/flan-t5-xxl` (note `xxl` vs. `xl`).

In[3]:

```
from langchain import HuggingFaceHub, LLMChain

# initialize Hub LLM
hub_llm = HuggingFaceHub(
repo_id='google/flan-t5-xl',
model_kwargs={'temperature':1e-10}
)

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

```
llm=hub_llm
)

# ask the user question about NFL 2010
print(llm_chain.run(question))
```

CopyOut[3]:

```
green bay packers
```

For this question, we get the correct answer of `"green bay packers"` .

Asking Multiple Questions

If we'd like to ask multiple questions, we can try two approaches:

1. Iterate through all questions using the `generate` method, answering them one at a time.
2. Place all questions into a single prompt for the LLM; this will only work for more advanced LLMs.

Starting with option (1), let's see how to use the `generate` method:

In[4]:

```
qs = [
{'question': "Which NFL team won the Super Bowl in the 2010 season?"},
```

```
{'question': "If I am 6 ft 4 inches, how tall am I in centimeters?"},
{'question': "Who was the 12th person on the moon?"},
{'question': "How many eyes does a blade of grass have?"}
]
res = llm_chain.generate(qs)
res
```

CopyOut[4]:

```
LLMResult(generations=[[Generation(text='green bay packers', generation_info=None)], [Generation(text='184',
generation_info=None)], [Generation(text='john glenn', generation_info=None)], [Generation(text='one',
generation_info=None)]], llm_output=None)
```

Here we get bad results except for the first question. This is simply a limitation of the LLM being used.

If the model cannot answer individual questions accurately, grouping all queries into a single prompt is unlikely to work. However, for the sake of experimentation, let's try it.

In[6]:

```
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=flan_t5
)

qs_str = (
```

```
"Which NFL team won the Super Bowl in the 2010 season?\n" +
"If I am 6 ft 4 inches, how tall am I in centimeters?\n" +
"Who was the 12th person on the moon?" +
"How many eyes does a blade of grass have?"
)

print(llm_chain.run(qs_str))
```

CopyOut[6]:

```
If I am 6 ft 4 inches, how tall am I in centimeters
```

As expected, the results are not helpful. We'll see later that more powerful LLMs can do this.

OpenAI LLMs

The OpenAI endpoints in LangChain connect to OpenAI directly or via Azure. We need an OpenAI account and API key to use these endpoints.

Once you have an API key, we add it to the `OPENAI_API_TOKEN` environment variable. We can do this with Python like so:

```
import os

os.environ['OPENAI_API_TOKEN'] = 'OPENAI_API_KEY'
```

Copy

Next, we must install the `openai` library via Pip.

```
!pip install openai
```

Now we can generate text using OpenAI's GPT-3 generation (or *completion*) models. We'll use `text-davinci-003` .

```
from langchain.llms import OpenAI

davinci = OpenAI(model_name='text-davinci-003')
```

Copy

Alternatively, if you're using OpenAI via Azure, you can do:

```
from langchain.llms import AzureOpenAI

llm = AzureOpenAI(
deployment_name="your-azure-deployment",
model_name="text-davinci-003"
)
```

Copy

We'll use the same simple question-answer prompt template as before with the Hugging Face example. The only change is that we now pass our OpenAI LLM `davinci`:

In[15]:

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

print(llm_chain.run(question))
```

CopyOut[15]:

```
The Green Bay Packers won the Super Bowl in the 2010 season.
```

As expected, we're getting the correct answer. We can do the same for multiple questions using `generate`:

In[16]:

```
qs = [
{'question': "Which NFL team won the Super Bowl in the 2010 season?"},
{'question': "If I am 6 ft 4 inches, how tall am I in centimeters?"},
{'question': "Who was the 12th person on the moon?"},
{'question': "How many eyes does a blade of grass have?"}
]
llm_chain.generate(qs)
```

CopyOut[16]:

LLMResult(generations=[[Generation(text=' The Green Bay Packers won the Super Bowl in the 2010 season.', generation_info={'finish_reason': 'stop', 'logprobs': None})], [Generation(text=' 193.04 centimeters', generation_info={'finish_reason': 'stop', 'logprobs': None})], [Generation(text=' Charles Duke was the 12th person on the moon. He was part of the Apollo 16 mission in 1972.', generation_info={'finish_reason': 'stop', 'logprobs': None})], [Generation(text=' A blade of grass does not have any eyes.', generation_info={'finish_reason': 'stop', 'logprobs': None})]], llm_output={'token_usage': {'total_tokens': 124, 'prompt_tokens': 75, 'completion_tokens': 49}})

Most of our results are correct or have a degree of truth. The model undoubtedly functions better than the `google/flan-t5-xl` model. As before, let's try feeding all questions into the model at once.

In[17]:

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

qs_str = (
"Which NFL team won the Super Bowl in the 2010 season?\n" +
"If I am 6 ft 4 inches, how tall am I in centimeters?\n" +
"Who was the 12th person on the moon?" +
"How many eyes does a blade of grass have?"
)

print(llm_chain.run(qs_str))
```

CopyOut[17]:

The New Orleans Saints won the Super Bowl in the 2010 season.

6 ft 4 inches is 193 centimeters.

The 12th person on the moon was Harrison Schmitt.

A blade of grass does not have eyes.

As we keep rerunning the query, the model will occasionally make errors, but at other times manage to get all answers correct.

That's it for our introduction to LangChain — a library that allows us to build more advanced apps around LLMs like OpenAI's GPT-3 models or the open-source alternatives available via Hugging Face.

As mentioned, LangChain can do much more than we've demonstrated here. We'll be covering these other features in upcoming articles.

References

[1] GPT-3 Archived Repo (2020), OpenAI GitHub

Share via:

Next

Prompt Templates and the Art of Prompts

LangChain AI Handbook

Chapters

1. **An Introduction to LangChain**
2. Prompt Templates and the Art of Prompts
3. Conversational Memory
4. Fixing Hallucination with Knowledge Bases
5. AI Agents
6. Custom Tools