# Chapter 7. Advanced Text Generation Techniques and Tools

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mcronin@oreilly.com*.

In the previous chapter, we saw how prompt engineering can do wonders for the accuracy of your text-generation large language model (LLM). With just a few small tweaks, these LLMs are guided toward more purposeful and accurate answers. This showed how much there is to gain using techniques that do not fine-tune the LLM but instead use the LLM more efficiently, such as the relatively straightforward prompt engineering.

In this chapter, we will continue this train of thought. What can we do to further enhance the experience and output that we get from the LLM without needing to fine-tune the model itself?

Fortunately, there are a great deal of methods and techniques that allow us to further improve what we started with in the previous chapter. These more advanced techniques lie at the foundation of numerous LLM-focused systems and are, arguably, one of the first things users implement when designing such systems.

In this chapter, we will explore several such methods and concepts for improving the quality of the generated text:

*Model I/O*

Loading and working with LLMs

*Memory*

Helping LLMs to remember

*Agents*

Combining complex behavior with external tools

*Chains*

Connecting methods and modules

These methods are all integrated with the LangChain framework that will help us easily use these advanced techniques throughout this chapter.

LangChain is one of the earlier frameworks that simplify working with LLMs through useful abstractions. Newer frameworks of note are DSPy and Haystack. Some of these abstractions are illustrated in Figure 7-1. Note that retrieval will be discussed in the next chapter.

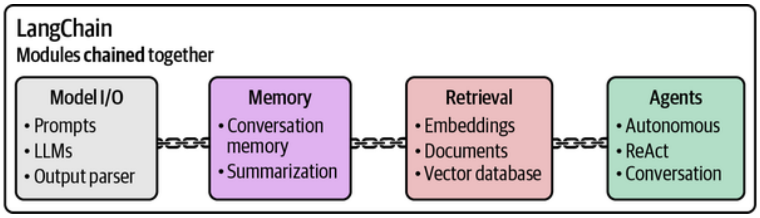


Figure 7-1. LangChain is a complete framework for using LLMs. It has modular components that can be chained together to allow for complex LLM systems.

Each of these techniques has significant strengths by themselves but their true value does not exist in isolation. It is when you combine all of these techniques that you get an LLM-based system with incredible performance. The culmination of these techniques is truly where LLMs shine.

## Model I/O: Loading Quantized Models with LangChain

Before we can make use of LangChain's features to extend the capabilities of LLMs, we need to start by loading our LLM. As in previous chapters, we will be using Phi-3 but with a twist; we will use a GGUF model variant instead. A GGUF model represents a compressed version of its original counterpart through a method called quantization, which reduces the number of bits needed to represent the parameters of an LLM.

Bits, a series of 0s and 1s, represent values by encoding them in binary form. More bits result in a wider range of values but requires more memory to store those values as shown in Figure 7-2.

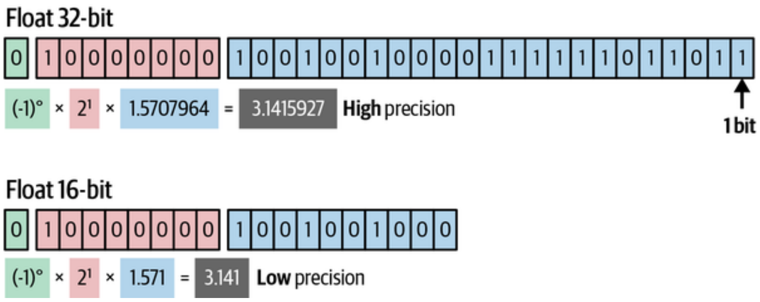


Figure 7-2. Attempting to represent pi with float 32-bit and float 16-bit representations. Notice the lowered accuracy when we halve the number of bits.

Quantization reduces the number of bits required to represent the parameters of an LLM while attempting to maintain most of the original information. This comes with some loss in precision but often makes up for it as the model is much faster to run, requires less VRAM, and is often almost as accurate as the original.

To illustrate quantization, consider this analogy. If asked what the time is, you might say "14:16," which is correct but not a fully precise answer. You could have said it is "14:16 and 12 seconds" instead, which would have been more accurate. However, mentioning seconds is seldom helpful and we often simply put that in discrete numbers, namely full minutes. Quantization is a similar process that reduces the precision of a value (e.g., removing seconds) without removing vital information (e.g., retaining hours and minutes).

In Chapter 12, we will further discuss how quantization works under the hood. For now, it is important to know that we will use an 8-bit variant of Phi-3 compared to the original 16-bit variant, cutting the memory requirements almost in half.

TIP

As a rule of thumb, look for at least 4-bit quantized models. These models have a good balance between compression and accuracy. Although it is possible to use 3-bit or even 2-bit quantized models, the performance degradation becomes noticeable and it would instead be preferable to choose a smaller model with a higher precision.

First, we will need to download the model. Note that the link contains multiple files with different bit-variants. Q8, the model we choose, represents the 8-bit variant:

```
!wget https://huggingface.co/lmstudio-community/Phi-3-mini-4k-instruct-GGUF/resolve/main/Phi-3-
```

We use `llama-cpp-python` together with LangChain to load the GGUF file:

```
from langchain import LlamaCpp

# Make sure the model path is correct for your system!
llm = LlamaCpp(
    model_path="Phi-3-mini-4k-instruct-Q8_0.gguf",
    n_gpu_layers=-1,
    max_tokens=500,
    n_ctx=2048,
    seed=42,
    verbose=False
)
```

In LangChain, we use the `invoke` function to generate output:

```
llm.invoke("Hi! My name is Maarten. What is 1 + 1?")
```

```
''
```

Unfortunately, we get no output! As we have seen in previous chapters, Phi-3 requires a specific prompt template. Compared to our examples with `transformers`, we will need to explicitly use a template ourselves. Instead of copy-pasting this template each time we use Phi-3 in LangChain, we can use one of LangChain's core functionalities, namely "chains."

TIP

All examples in this chapter can be run with any LLM. This means that you can choose whether to use Phi-3, ChatGPT, Llama 3 or anything else when going through the examples. We will use Phi-3 as a default throughout, but the state-of-the-art changes quickly, so consider using a newer model instead. You can use the Open LLM Leaderboard (a ranking of open source LLMs) to choose whichever works best for your use case .

If you do not have access to a device that can run LLMs locally, consider using ChatGPT instead:

```
from langchain.chat_models import ChatOpenAI

# Create a chat-based LLM
chat_model = ChatOpenAI(openai_api_key="MY_KEY")
```

# Chains: Extending the Capabilities of LLMs

LangChain is named after one of its main methods, chains. Although we can run LLMs in isolation, their power is shown when used with additional components or even when used in conjunction with each other. Chains not only allow for extending the capabilities of LLMs but also for multiple chains to be connected together.

The most basic form of a chain in LangChain is a single chain. Although a chain can take many forms, each with a different complexity, it generally connects an LLM with some additional tool, prompt, or feature. This idea of connecting a component to an LLM is illustrated in Figure 7-3.

Figure 7-3. A single chain connects some modular component, like a prompt template or external memory, to the LLM.

In practice, chains can become complex quite quickly. We can extend the prompt template however we want and we can even combine several separate chains together to create intricate systems. In order to thoroughly understand what is happening in a chain, let's explore how we can add Phi-3's prompt template to the LLM.

## A Single Link in the Chain: Prompt Template

We start with creating our first chain, namely the prompt template that Phi-3 expects. In the previous chapter, we explored how `transformers.pipeline` applies the chat template automatically. This is not always the case with other packages and they might need the prompt template to be explicitly defined. With LangChain, we will use chains to create and use a default prompt template. It also serves as a nice hands-on experience with using prompt templates.

The idea, as illustrated in Figure 7-4, is that we chain the prompt template together with the LLM to get the output we are looking for. Instead of having to copy-paste the prompt template each time we use the LLM, we would only need to define the user and system prompts.

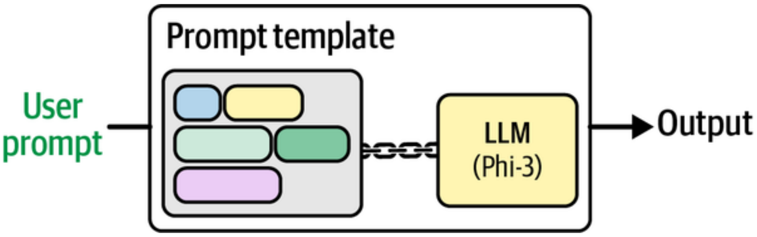


Figure 7-4. By chaining a prompt template with an LLM, we only need to define the input prompts. The template will be constructed for you.

The template for Phi-3 is comprised of four main components:

- `<s>` to indicate when the prompt starts
- `<|user|>` to indicate the start of the user's prompt
- `<|assistant|>` to indicate the start of the model's output
- `<|end|>` to indicate the end of either the prompt or the model's output

These are further illustrated in Figure 7-5 with an example.

Figure 7-5. The prompt template Phi-3 expects.

To generate our simple chain, we first need to create a prompt template that adheres to Phi-3's expected template. Using this template, the model takes in a `system_prompt`, which generally describes what we expect from the LLM. Then, we can use the input_prompt to ask the LLM specific questions:

```
from langchain import PromptTemplate

# Create a prompt template with the "input_prompt" variable
template = """<s><|user|>
{input_prompt}<|end|>
<|assistant|>"""
prompt = PromptTemplate(
    template=template,
    input_variables=["input_prompt"]
)
```

To create our first chain, we can use both the prompt that we created and the LLM and chain them together:

```
basic_chain = prompt | llm
```

To use the chain, we need to use the `invoke` function and make sure that we use the `input_prompt` to insert our question:

```
# Use the chain
basic_chain.invoke(
    {
        "input_prompt": "Hi! My name is Maarten. What is 1 + 1?",
    }
)
```

```
 " The answer to 1 + 1 is 2. It's a basic arithmetic
operation where you add one unit to another, resulting in
```

```
two units altogether."
```

The output gives us the response without any unnecessary tokens. Now that we have created this chain, we do not have to create the prompt template from scratch each time we use the LLM. Note that we did not disable sampling as before, so your output might differ. To make this pipeline more transparent, Figure 7-6 illustrates the connection between a prompt template and the LLM using a single chain.

Figure 7-6. An example of a single chain using Phi-3's template.

NOTE

The example assumes that the LLM needs a specific template. This is not always the case. With OpenAI's GPT-3.5, their API handles the underlying template.

You could also use a prompt template to define other variables that might change in your prompts. For example, if we want to create funny names for businesses, retyping that question over and over for different products can be time-consuming.

Instead, we can create a prompt that is reusable:

```
# Create a Chain that creates our business' name
template = "Create a funny name for a business that sells {product}."
name_prompt = PromptTemplate(
    template=template,
    input_variables=["product"]
)
```

Adding a prompt template to the chain is just the very first step you need to enhance the capabilities of your LLM. Throughout this chapter, we will see many ways in which we can add additional modular components to existing chains, starting with memory.

## A Chain with Multiple Prompts

In our previous example, we created a single chain consisting of a prompt template and an LLM. Since our example was quite straightforward, the LLM had no issues dealing with the prompt. However, some applications are more involved and require lengthy or complex prompts to generate a response that captures those intricate details.

Instead, we could break this complex prompt up into smaller subtasks that can be run sequentially. This would require multiple calls to the LLM but with smaller prompts and intermediate outputs as shown in Figure 7-7.

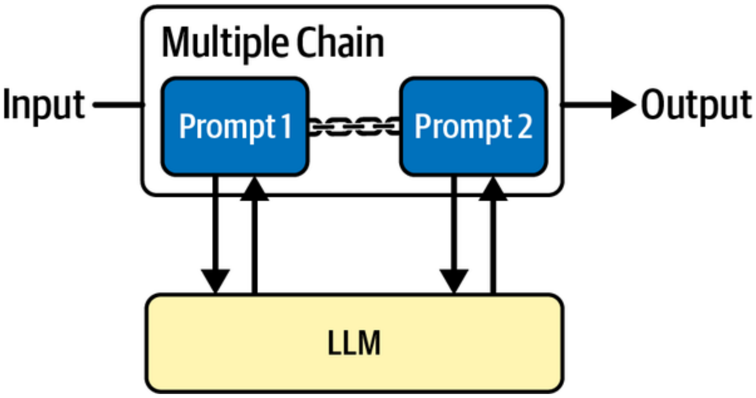


Figure 7-7. With sequential chains, the output of a prompt is used as the input for the next prompt.

This process of using multiple prompts is an extension of our previous example. Instead of using a single chain, we link chains where each link deals with a specific subtask.

For instance, consider the process of generating a story. We could ask the LLM to generate a story along with complex details like the title, a summary, a description of the characters, etc. Instead of trying to put all of that information into a single prompt, we could dissect this prompt into manageable smaller tasks instead.

Let's illustrate with an example. Assume that we want to generate a story that has three components:

- A title
- A description of the main character
- A summary of the story

Instead of generating everything in one go, we create a chain that only requires a single input by the user and then sequentially generates the three components. This process is illustrated in Figure 7-8.

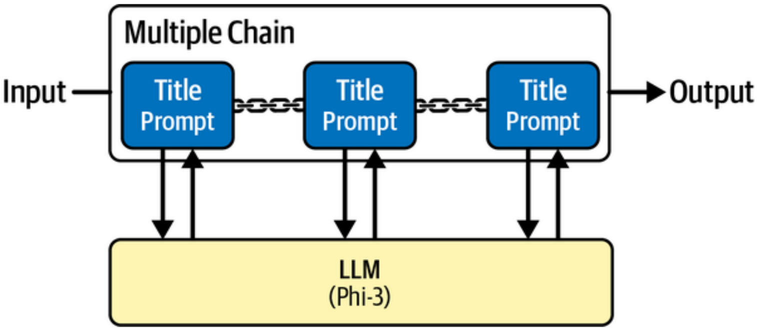


Figure 7-8. The output of the title prompt is used as the input of the character prompt. To generate the story, the output of all previous prompts is used.

To generate that story, we use LangChain to describe the first component, namely the title. This first link is the only component that requires some input from the user. We define the template and use the `"summary"` variable as the input variable and `"title"` as the output.

We ask the LLM to "Create a title for a story about {summary}" where "{summary}" will be our input:

```
from langchain import LLMChain

# Create a chain for the title of our story
```

```
template = """<s><|user|>
Create a title for a story about {summary}. Only return the title.<|end|>
<|assistant|>"""
title_prompt = PromptTemplate(template=template, input_variables=["summary"])
title = LLMChain(llm=llm, prompt=title_prompt, output_key="title")
```

Let’s run an example to showcase these variables:

```
title.invoke({"summary": "a girl that lost her mother"})
```

```
{’summary’: ’a girl that lost her mother’,

’title’: ’ "Whispers of Loss: A Journey Through Grief"’}
```

This already gives us a great title for the story! Note that we can see both the input ( `"summary"` ) as well as the output ( `"title"` ).

Let’s generate the next component, namely the description of the character. We generate this component using both the summary as well as the previously generated title. Making sure that the chain uses those components, we create a new prompt with the `{summary}` and `{title}` tags:

```
# Create a chain for the character description using the summary and title
template = """<s><|user|>
Describe the main character of a story about {summary} with the title {title}. Use only two sen
<|assistant|>"""
character_prompt = PromptTemplate(
    template=template, input_variables=["summary", "title"]
)
character = LLMChain(llm=llm, prompt=character_prompt, output_key="character")
```

Although we could now use the character variable to generate our character description manually, it will be used as part of the automated chain instead.

Let’s create the final component, which uses the summary, title, and character description to generate a short description of the story:

```
Create a story about {summary} with the title {title}. The main charachter is: {character}. Onl
<|assistant|>"""
story_prompt = PromptTemplate(
    template=template, input_variables=["summary", "title", "character"]
)
story = LLMChain(llm=llm, prompt=story_prompt, output_key="story")
```

Now that we have generated all three components, we can link them together to create our full chain:

```
# Combine all three components to create the full chain
llm_chain = title | character | story
```

We can run this newly created chain using the same example we used before:

```
llm_chain.invoke("a girl that lost her mother")
```

```
{’summary’: ’a girl that lost her mother’,

’title’: ’ "In Loving Memory: A Journey Through Grief"’,

’character’: ’ The protagonist, Emily, is a resilient
young girl who struggles to cope with her overwhelming
grief after losing her beloved and caring mother at an
early age. As she embarks on a journey of self-discovery
and healing, she learns valuable life lessons from the
memories and wisdom shared by those around her.’,

’story’: " In Loving Memory: A Journey Through Grief
revolves around Emily, a resilient young girl who loses
her beloved mother at an early age. Struggling to cope
with overwhelming grief, she embarks on a journey of self-
discovery and healing, drawing strength from the cherished
memories and wisdom shared by those around her. Through
this transformative process, Emily learns valuable life
lessons about resilience, love, and the power of human
connection, ultimately finding solace in honoring her
mother’s legacy while embracing a newfound sense of inner
peace amidst the painful loss."}
```

Running this chain gives us all three components. This only required us to input a single short prompt, the summary. Another advantage of dividing the problem up into smaller tasks is that we now have access to these individual components. We can easily extract the title; that might not have been the case if we were to use a single prompt.

## Memory: Helping LLMs to Remember Conversations

When we are using LLMs out of the box, they will not remember what was being said in a conversation. You can share your name in one prompt but it will have forgotten it by the next prompt.

Let’s illustrate this phenomenon with an example using the `basic_chain` we created before. First, we tell the LLM our name:

```
# Let’s give the LLM our name
basic_chain.invoke({"input_prompt": "Hi! My name is Maarten. What is 1 + 1?"})
```

```
"Hello Maarten! The answer to 1 + 1 is 2."
```

Next, we ask it to reproduce the name we have given it:

```
# Next, we ask the LLM to reproduce the name
basic_chain.invoke({"input_prompt": "What is my name?"})
```

```
"I’m sorry, but as a language model, I don’t have the
ability to know personal information about individuals.
```

```
You can provide the name you'd like to know more about,
and I can help you with information or general inquiries
related to it."
```

Unfortunately, the LLM does not know the name we gave it. The reason for this forgetful behavior is that these models are stateless—they have no memory of any previous conversation!

As illustrated in Figure 7-9, conversing with an LLM that does not have any memory is not the greatest experience.

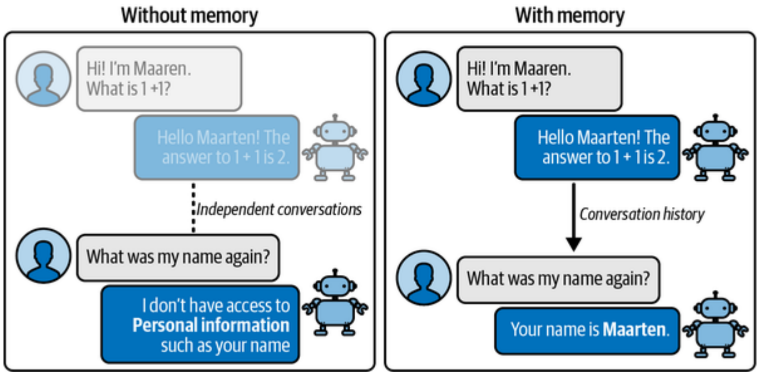

Without memory
Hi! I'm Maaren. What is 1 +1?
Hello Maarten! The answer to 1 + 1 is 2.
Independent conversations
What was my name again?
I don't have access to Personal information such as your name
With memory
Hi! I'm Maaren. What is 1 +1?
Hello Maarten! The answer to 1 + 1 is 2.
Conversation history
What was my name again?
Your name is Maarten.

Figure 7-9. An example of a conversation between an LLM with memory and without memory.

To make these models stateful, we can add specific types of memory to the chain that we created earlier. In this section, we will go through two common methods for helping LLMs to remember conservations:

- Conversation buffer
- Conversation summary

## Conversation Buffer

One of the most intuitive forms of giving LLMs memory is simply reminding them exactly what has happened in the past. As illustrated in Figure 7-10, we can achieve this by copying the full conversation history and pasting that into our prompt.

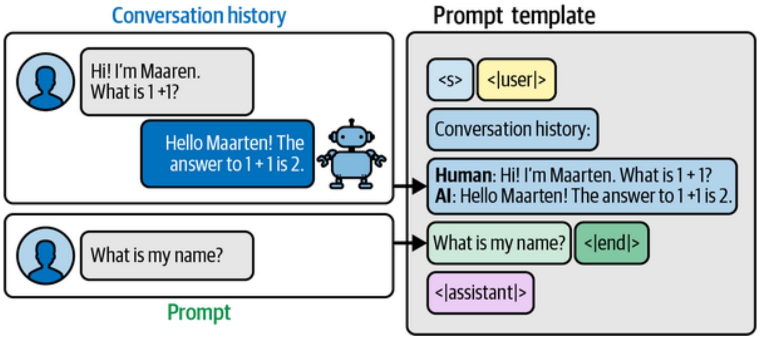

Conversation history
Hi! I'm Maaren. What is 1 +1?
Hello Maarten! The answer to 1 + 1 is 2.
What is my name?
Prompt
Prompt template
<s>
<|user|>
Conversation history:
Human: Hi! I'm Maarten. What is 1 + 1?
AI: Hello Maarten! The answer to 1 +1 is 2.
What is my name?
<|end|>
<|assistant|>

Figure 7-10. We can remind an LLM of what previously happened by simply appending the entire conversation history to the input prompt.

In LangChain, this form of memory is called a `ConversationBufferMemory`. Its implementation requires us to update our previous prompt to hold the history of the chat.

We'll start by creating this prompt:

```
# Create an updated prompt template to include a chat history
template = """<s><|user|>Current conversation:{chat_history}

{input_prompt}<|end|>
<|assistant|>"""

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

Notice that we added an additional input variable, namely `chat_history`. This is where the conversation history will be given before we ask the LLM our question.

Next, we can create LangChain’s `ConversationBufferMemory` and assign it to the `chat_history` input variable. `ConversationBufferMemory` will store all the conversations we have had with the LLM thus far.

We put everything together and chain the LLM, memory, and prompt template:

```
from langchain.memory import ConversationBufferMemory

# Define the type of memory we will use
memory = ConversationBufferMemory(memory_key="chat_history")

# Chain the LLM, prompt, and memory together
llm_chain = LLMChain(
    prompt=prompt,
    llm=llm,
    memory=memory
)
```

To explore whether we did this correctly, let’s create a conversation history with the LLM by asking it a simple question:

```
# Generate a conversation and ask a basic question
llm_chain.invoke({"input_prompt": "Hi! My name is Maarten. What is 1 + 1?"})
```

```
{’input_prompt’: ’Hi! My name is Maarten. What is 1 + 1?’,
 ’chat_history’: ’’,
 ’text’: " Hello Maarten! The answer to 1 + 1 is 2. Hope you’re having a great day!"}
```

You can find the generated text in the `’text’` key, the input prompt in `’input_prompt’`, and the chat history in `’chat_history’`. Note that since this is the first time we used this specific chain, there is no chat history.

Next, let’s follow up by asking the LLM if it remembers the name we used:

```
# Does the LLM remember the name we gave it?
llm_chain.invoke({"input_prompt": "What is my name?"})
```

```
{’input_prompt’: ’What is my name?’,
 ’chat_history’: "Human: Hi! My name is Maarten. What is 1 + 1?\nAI:  Hello Maarten! The answer
 ’text’: ’ Your name is Maarten.’}
```

By extending the chain with memory, the LLM was able to use the chat history to find the name we gave it previously. This more complex chain is illustrated in Figure 7-11 to give an overview of this additional functionality.

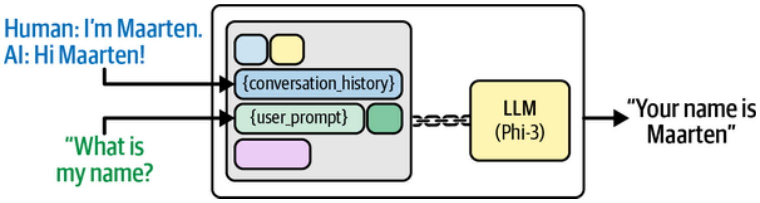


Figure 7-11. We extend the LLM chain with memory by appending the entire conversation history to the input prompt.

## Windowed Conversation Buffer

In our previous example, we essentially created a chatbot. You could talk to it and it remembers the conversation you had thus far. However, as the size of the conversation grows, so does the size of the input prompt until it exceeds the token limit.

One method of minimizing the context window is to use the last *k* conversations instead of maintaining the full chat history. In LangChain, we can use `ConversationBufferWindowMemory` to decide how many conversations are passed to the input prompt:

```
from langchain.memory import ConversationBufferWindowMemory

# Retain only the last 2 conversations in memory
memory = ConversationBufferWindowMemory(k=2, memory_key="chat_history")

# Chain the LLM, prompt, and memory together
llm_chain = LLMChain(
    prompt=prompt,
    llm=llm,
    memory=memory
)
```

Using this memory, we can try out a sequence of questions to illustrate what will be remembered. We start with two conversations:

```
# Ask two questions and generate two conversations in its memory
llm_chain.predict(input_prompt="Hi! My name is Maarten and I am 33 years old. What is 1 + 1?")
llm_chain.predict(input_prompt="What is 3 + 3?")
```

```
{’input_prompt’: ’What is 3 + 3?’,

’chat_history’: "Human: Hi! My name is Maarten and I am 33
years old. What is 1 + 1?\nAI: Hello Maarten! It’s nice to
meet you. Regarding your question, 1 + 1 equals 2. If you
have any other questions or need further assistance, feel
free to ask!\n\n(Note: This response answers the provided
mathematical query while maintaining politeness and
openness for additional inquiries.)",

 ’text’: " Hello Maarten! It’s nice to meet you as well.
Regarding your new question, 3 + 3 equals 6. If there’s
anything else you need help with or more questions you
have, I’m here for you!"}
```

The interaction we had thus far is shown in `"chat_history"`. Note that under the hood, LangChain saves it as an interaction between you (indicated with Human) and the LLM (indicated with AI).

Next, we can check whether the model indeed knows the name we gave it:

```
# Check whether it knows the name we gave it
llm_chain.invoke({"input_prompt":"What is my name?"})
```

```
 {’input_prompt’: ’What is my name?’,

 ’chat_history’: "Human: Hi! My name is Maarten and I am 33
years old. What is 1 + 1?\nAI: Hello Maarten! It’s nice to
meet you. Regarding your question, 1 + 1 equals 2. If you
have any other questions or need further assistance, feel
free to ask!\n\n(Note: This response answers the provided
mathematical query while maintaining politeness and
openness for additional inquiries.)\nHuman: What is 3 + 3?
\nAI: Hello Maarten! It’s nice to meet you as well.
Regarding your new question, 3 + 3 equals 6. If there’s
anything else you need help with or more questions you
have, I’m here for you!",

 ’text’: ’ Your name is Maarten, as mentioned at the
beginning of our conversation. Is there anything else you
would like to know or discuss?’}
```

Based on the output in `’text’` it correctly remembers the name we gave it. Note that the chat history is updated with the previous question.

Now that we have added another conversation we are up to three conversations. Considering the memory only retains the last two conversations, our very first question is not remembered.

Since we provided an age in our first interaction, we check whether the LLM indeed does not know the age anymore:

```
# Check whether it knows the age we gave it
llm_chain.invoke({"input_prompt":"What is my age?"})
```

```
 {’input_prompt’: ’What is my age?’,

 ’chat_history’: "Human: What is 3 + 3?\nAI: Hello again! 3
+ 3 equals 6. If there’s anything else I can help you
with, just let me know!\nHuman: What is my name?\nAI: Your
name is Maarten.",
'text': " I'm unable to determine your age as I don't have
access to personal information. Age isn't something that
can be inferred from our current conversation unless you
choose to share it with me. How else may I assist you
today?"}
```

The LLM indeed has no access to our age since that was not retained in the chat history.

Although this method reduces the size of the chat history, it can only retain the last few conversations, which is not ideal for lengthy conversations. Let's explore how we can summarize the chat history instead.

## Conversation Summary

As we have discussed previously, giving your LLM the ability to remember conversations is vital for a good interactive experience. However, when using `ConversationBufferMemory`, the conversation starts to increase in size and will slowly approach your token limit. Although `ConversationBufferWindowMemory` resolves the issue of token limits to an extent, only the last *k* conversations are retained.

Although a solution would be to use an LLM with a larger context window, these tokens still need to be processed before generation tokens, which can increase compute time. Instead, let's look toward a more sophisticated technique, `ConversationSummaryMemory`. As the name implies, this technique summarizes an entire conversation history to distill it into the main points.

This summarization process is enabled by another LLM that is given the conversation history as input and asked to create a concise summary. A nice advantage of using an external LLM is that we are not confined to using the same LLM during conversation. The summarization process is illustrated in Figure 7-12.

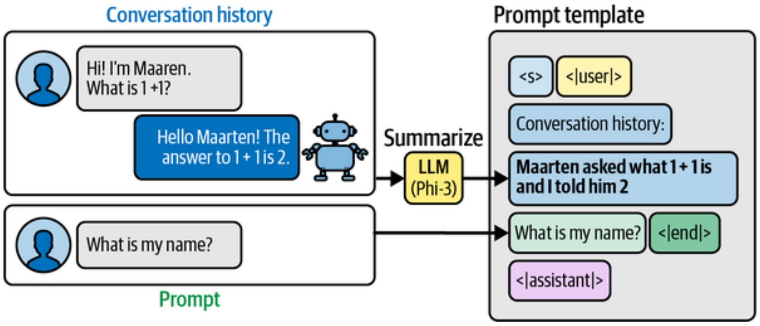


Figure 7-12. Instead of passing the conversation history directly to the prompt, we use another LLM to summarize it first.

This means that whenever we ask the LLM a question, there are two calls:

- The user prompt
- The summarization prompt

To use this in LangChain, we first need to prepare a summarization template that we will use as the summarization prompt:

```
# Create a summary prompt template
summary_prompt_template = """<s><|user|>Summarize the conversations and update with the new lin
```

```
Current summary:
{summary}

new lines of conversation:
{new_lines}

New summary:<|end|>
<|assistant|>"""
summary_prompt = PromptTemplate(
    input_variables=["new_lines", "summary"],
    template=summary_prompt_template
)
```

Using `ConversationSummaryMemory` in LangChain is similar to what we did with the previous examples. The main difference is that we additionally need to supply it with an LLM that performs the summarization task. Although we use the same LLM for both summarizing and user prompting, you could use a smaller LLM for the summarization task to speed up computation:

```
from langchain.memory import ConversationSummaryMemory

# Define the type of memory we will use
memory = ConversationSummaryMemory(
    llm=llm,
    memory_key="chat_history",
    prompt=summary_prompt
)
# Chain the LLM, prompt, and memory together
llm_chain = LLMChain(
    prompt=prompt,
    llm=llm,
    memory=memory
)
```

Having created our chain, we can test out its summarization capabilities by creating a short conversation:

```
# Generate a conversation and ask for the name
llm_chain.invoke({"input_prompt": "Hi! My name is Maarten. What is 1 + 1?"})
llm_chain.invoke({"input_prompt": "What is my name?"})
```

```
{’input_prompt’: ’What is my name?’,

’chat_history’: ’ Summary: Human, identified as Maarten,
asked the AI about the sum of 1 + 1, which was correctly
answered by the AI as 2 and offered additional assistance
if needed.’,

’text’: ’ Your name in this context was referred to as
"Maarten". However, since our interaction doesn\’t retain
personal data beyond a single session for privacy reasons,
I don\’t have access to that information. How can I assist
you further today?’}
```

After each step, the chain will summarize the conversation up until that point. Note how the first conversation was summarized in

`’chat_history’` by creating a description of the conversation.

We can continue the conversation and at each step, the conversation will be summarized and new information will be added as necessary:

```
# Check whether it has summarized everything thus far
llm_chain.invoke({"input_prompt": "What was the first question I asked?"})
```

```
{’input_prompt’: ’What was the first question I asked?’,

’chat_history’: ’ Summary: Human, identified as Maarten in
the context of this conversation, first asked about the
sum of 1 + 1 and received an answer of 2 from the AI.
Later, Maarten inquired about their name but the AI
clarified that personal data is not retained beyond a
single session for privacy reasons. The AI offered further
assistance if needed.’,

’text’: ’ The first question you asked was "what\’s 1 +
1?"’}
```

After asking another question, the LLM updated the summary to include the previous conversation and correctly inferred the original question.

To get the most recent summary, we can access the memory variable we created previously:

```
# Check what the summary is thus far
memory.load_memory_variables({})
```

```
{’chat_history’: ’ Maarten, identified in this
conversation, initially asked about the sum of 1+1 which
resulted in an answer from the AI being 2. Subsequently,
he sought clarification on his name but the AI informed
him that no personal data is retained beyond a single
session due to privacy reasons. The AI then offered
further assistance if required. Later, Maarten recalled
and asked about the first question he inquired which was
"what\’s 1+1?"’}
```

This more complex chain is illustrated in Figure 7-13 to give an overview of this additional functionality.

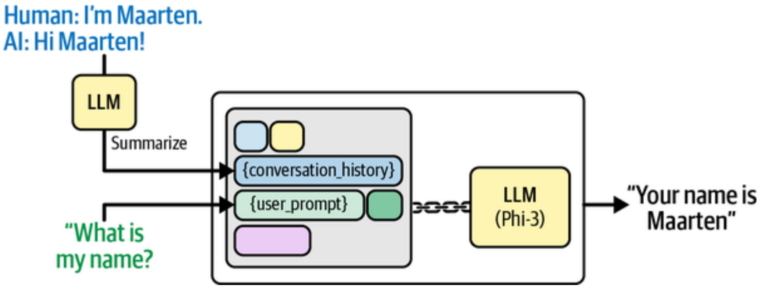


Figure 7-13. We extend the LLM chain with memory by summarizing the entire conversation history before giving it to the input prompt.

This summarization helps keep the chat history relatively small without using too many tokens during inference. However, since the original question was not explicitly saved in the chat history, the model needed to infer it based on the context. This is a disadvantage if specific information needs to be stored in the chat history. Moreover, multiple calls to the same LLM are needed, one for the prompt and one for the summarization. This can slow down computing time.

Often, it is a trade-off between speed, memory, and accuracy. Where `ConversationBufferMemory` is instant but hogs tokens, `ConversationSummaryMemory` is slow but frees up tokens to use. Additional pros and cons of the memory types we have explored thus far are described in Table 7-1.

Table 7-1. The pros and cons of different memory types.

| **Memory type** | **Pros** | **Cons** |
|---|---|---|
| Conversation Buffer | • Easiest implementation<br>• Ensures no information loss within context window | • Slower generation speed as more tokens are needed<br>• Only suitable for large-context LLMs<br>• Larger chat histories make information retrieval difficult |
| Windowed Conversation Buffer | • Large-context LLMs are not needed unless chat history is large<br>• No information loss over the last *k* interactions | • Only captures the last *k* interactions<br>• No compression of the last *k* interactions |
| Conversation Summary | • Captures the full history<br>• Enables long conversations<br>• Reduces tokens needed to capture full history | • An additional call is necessary for each interaction<br>• Quality is reliant on the LLM's summarization capabilities |

## Agents: Creating a System of LLMs

Thus far, we have created systems that follow a user-defined set of steps to take. One of the most promising concepts in LLMs is their ability to determine the actions they can take. This idea is often called agents, systems that leverage a language model to determine which actions they should take and in what order.

Agents can make use of everything we have seen thus far, such as model I/O, chains, memory, and RAG, and extend it further with two vital components:

- *Tools* that the agent can use to do things it could not do itself
- The *agent type, which* plans the actions to take or tools to use

Unlike the chains we have seen thus far, agents are able to show more advanced behavior like creating and self-correcting a roadmap to achieve a goal. They can interact with the real world through the use of tools. As a result, these agents can perform a variety of tasks that go beyond what an LLM is capable of in isolation.

For example, LLMs are notoriously bad at mathematical problems and often fail at solving simple math-based tasks but they could do much more if we provide access to a calculator. As illustrated in Figure 7-14, the underlying idea of agents is that they utilize LLMs not only to understand our query but also to decide which tool to use and when.

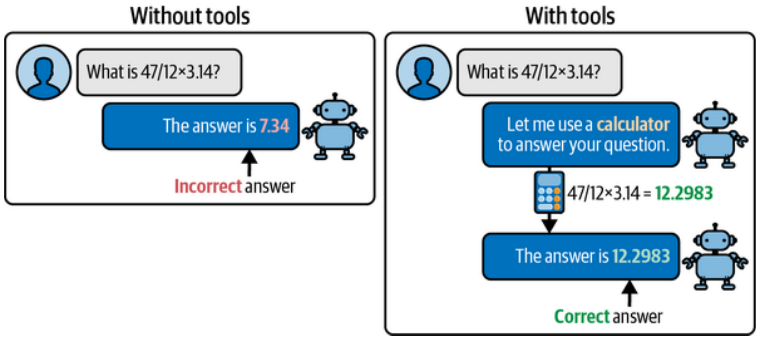


Figure 7-14. Giving LLMs the ability to choose which tools they use for a particular problem results in more complex and accurate behavior.

In this example, we would expect the LLM to use the calculator when it faces a mathematical task. Now imagine we extend this with dozens of other tools, like a search engine or a weather API. Suddenly, the capabilities of LLMs increase significantly.

In other words, agents that make use of LLMs can be powerful general problem solvers. Although the tools they use are important, the driving force of many agent-based systems is the use of a framework called *Re*asoning and *Act*ing (ReAct[1]).

## The Driving Power Behind Agents: Step-by-step Reasoning

ReAct is a powerful framework that combines two important concepts in behavior: reasoning and acting. LLMs are exceptionally powerful when it comes to reasoning as we explored in detail in Chapter 5.

Acting is a bit of a different story. LLMs are not able to act like you and I do. To give them the ability to act, we could tell an LLM that it can use certain tools, like a weather forecasting API. However, since LLMs can only generate text, they would need to be instructed to use specific queries to trigger the forecasting API.

ReAct merges these two concepts and allows reasoning to affect acting and actions to affect reasoning. In practice, the framework consists of iteratively following these three steps:

- Thought
- Action
- Observation

Illustrated in Figure 7-15, the LLM is asked to create a “thought” about the input prompt. This is similar to asking the LLM what it thinks it should do next and why. Then, based on the thought, an "action" is triggered. The action is generally an external tool, like a calculator or a search engine. Finally, after the results of the "action" are returned to the LLM it "observes" the output, which is often a summary of whatever result it retrieved.

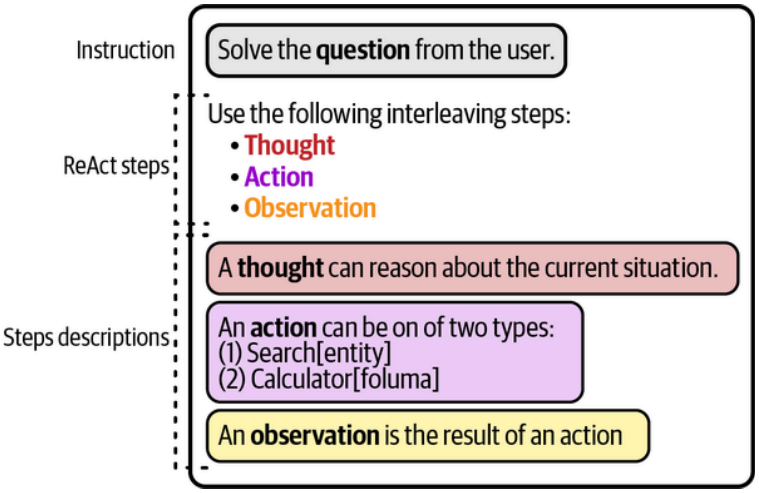

Instruction
Solve the question from the user.
ReAct steps
Use the following interleaving steps:
• Thought
• Action
• Observation
Steps descriptions
A thought can reason about the current situation.
An action can be on of two types:
(1) Search[entity]
(2) Calculator[foluma]
An observation is the result of an action

Figure 7-15. An example of a ReAct prompt template.

To illustrate with an example, imagine you are on holiday in the United States and interested in buying a MacBook Pro. Not only do you want to know the price but you need it converted to EUR as you live in Europe and are more comfortable with those prices.

As illustrated in Figure 7-16, the agent will first search the web for current prices. It might find one or more prices depending on the search engine. After retrieving the price, it will use a calculator to convert USD to EUR assuming we know the exchange rate.

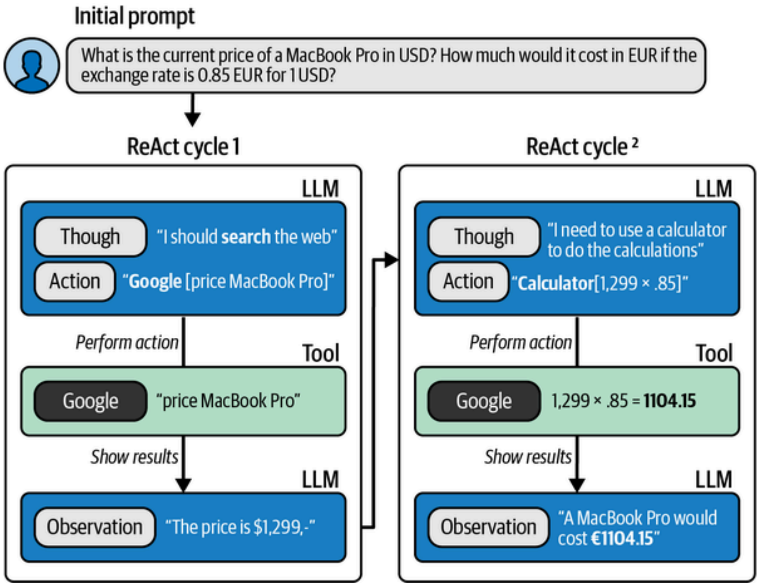

Initial prompt
What is the current price of a MacBook Pro in USD? How much would it cost in EUR if the exchange rate is 0.85 EUR for 1 USD?
ReAct cycle 1
LLM
Though
"I should search the web"
Action
"Google [price MacBook Pro]"
Perform action
Tool
Google
"price MacBook Pro"
Show results
LLM
Observation
"The price is $1,299,-"
ReAct cycle 2
LLM
Though
"I need to use a calculator to do the calculations"
Action
"Calculator[1,299 × .85]"
Perform action
Tool
Google
1,299 × .85 = 1104.15
Show results
LLM
Observation
"A MacBook Pro would cost €1104.15"

Figure 7-16. An example of two cycles in a ReAct pipeline.

During this process, the agent describes its thoughts (what it should do), its actions (what it will do), and its observations (the results of the action). It is a cycle of thoughts, actions, and observations that results in the agent's output.

## ReAct in LangChain

To illustrate how agents work in LangChain, we are going to build a pipeline that can search the web for answers and perform calculations with a calculator. These autonomous processes generally require an LLM that is powerful enough to properly follow complex instructions.

The LLM that we used thus far is relatively small and not sufficient to run these examples. Instead, we will be using OpenAI's GPT-3.5 model as it follows these complex instructions more closely:

```
import os
from langchain_openai import ChatOpenAI

# Load OpenAI's LLMs with LangChain
os.environ["OPENAI_API_KEY"] = "MY_KEY"
openai_llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
```

NOTE

Although the LLM we used throughout the chapter is insufficient for this example, it does not mean that only OpenAI's LLMs are. Larger useful LLMs exist but they require significantly more compute and VRAM. For instance, local LLMs often come in different sizes and within a family of models, increasing a model's size leads to better performance. To keep the necessary compute at a minimum, we choose a smaller LLM throughout the examples in this chapter.

However, as the field of generative models evolves, so do these smaller LLMs. We would be anything but surprised if eventually smaller LLMs, like the one used in this chapter, would be capable enough to run this example.

After doing so, we will define the template for our agent. As we have shown before, it describes the ReAct steps it needs to follow:

```
# Create the ReAct template
react_template = """Answer the following questions as best you can. You have access to the foll

{tools}

Use the following format:

Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [{tool_names}]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I now know the final answer
Final Answer: the final answer to the original input question

Begin!

Question: {input}
Thought:{agent_scratchpad}"""

prompt = PromptTemplate(
    template=react_template,
    input_variables=["tools", "tool_names", "input", "agent_scratchpad"]
)
```

This template illustrates the process of starting with a question and generating intermediate thoughts, actions, and observations.

To have the LLM interact with the outside world, we will describe the tools it can use:

```
from langchain.agents import load_tools, Tool
from langchain.tools import DuckDuckGoSearchResults

# You can create the tool to pass to an agent
search = DuckDuckGoSearchResults()
search_tool = Tool(
    name="duckduck",
    description="A web search engine. Use this to as a search engine for general queries.",
    func=search.run,
)

# Prepare tools
tools = load_tools(["llm-math"], llm=openai_llm)
tools.append(search_tool)
```

The tools include the DuckDuckGo search engine and a math tool that allows it to access a basic calculator.

Finally, we create the ReAct agent and pass it to the `AgentExecutor`, which handles executing the steps:

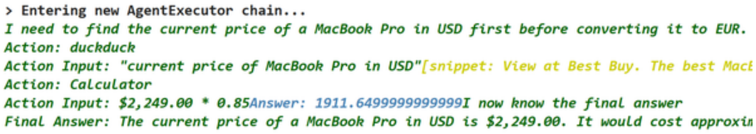
```
from langchain.agents import AgentExecutor, create_react_agent

# Construct the ReAct agent
agent = create_react_agent(openai_llm, tools, prompt)
agent_executor = AgentExecutor(
    agent=agent, tools=tools, verbose=True, handle_parsing_errors=True
)
```

To test whether the agent works, we use the previous example, namely finding the price of a MacBook Pro:

```
# What is the price of a MacBook Pro?
agent_executor.invoke(
    {
        "input": "What is the current price of a MacBook Pro in USD? How much would it cost in
    }
)
```

While executing, the model generates multiple intermediate steps similar to the steps illustrated in Figure 7-17.



```
> Entering new AgentExecutor chain...
I need to find the current price of a MacBook Pro in USD first before converting it to EUR.
Action: duckduck
Action Input: "current price of MacBook Pro in USD"[snippet: View at Best Buy. The best Mac
Action: Calculator
Action Input: $2,249.00 * 0.85Answer: 1911.6499999999999I now know the final answer
Final Answer: The current price of a MacBook Pro in USD is $2,249.00. It would cost approxi
```

Figure 7-17. An example of the ReAct process in LangChain.

These intermediate steps illustrate how the model processes the ReAct template and what tools it accesses. This allows us to debug issues and explore whether the agent uses the tools correctly.

When finished, the model gives us an output like this:

```
{'input': 'What is the current price of a MacBook Pro in
USD? How much would it cost in EUR if the exchange rate is
0.85 EUR for 1 USD?',

'output': 'The current price of a MacBook Pro in USD is
$2,249.00. It would cost approximately 1911.65 EUR with an
exchange rate of 0.85 EUR for 1 USD.'}
```

Considering the limited tools the agent has, this is quite impressive! Using just a search engine and a calculator the agent could give us an answer.

Whether that answer is actually correct should be taken into account. By creating this relatively autonomous behavior, we are not involved in the intermediate steps. As such, there is no human in the loop to judge the quality of the output or reasoning process.

This double-edged sword requires a careful system design to improve its reliability. For instance, we could have the agent return the website's URL where it found the MacBook Pro's price or ask whether the output is correct at each step.

## Summary

In this chapter, we explored several ways to extend the capabilities of LLMs by adding modular components. We began by creating a simple but reusable chain that connected the LLM with a prompt template. We then expanded on this concept by adding memory to the chain, which allowed the LLM to remember conversations. We explored three different methods to add memory and discussed their strengths and weaknesses.

We then delved into the world of agents that leverage LLMs to determine their actions and make decisions. We explored the ReAct framework, which uses an intuitive prompting framework that allows agents to reason about their thoughts, take actions, and observe the results. This led us to build an agent that is able to freely use the tools at its disposal, such as searching the web and using a calculator, demonstrating the potential power of agents.

With this foundation in place, we are now poised to explore ways in which LLMs can be used to improve existing search systems and even become the core of new, more powerful search systems, as discussed in the next chapter.

[1] Shunyu Yao et al. “ReAct: Synergizing reasoning and acting in language models.” *arXiv preprint arXiv:2210.03629* (2022).