第八章　メモリ

デフォルトでは、LLM（大規模言語モデル）やエージェントはすべて無状態であり、各モデルの呼び出しは他のインタラクションとは独立しています。つまり、ユーザーがAPIを通じて大規模言語モデルとの新しい対話を開始するたびに、モデルが実際には昨日や一昨日に行った対話のことを知ることはありません。

しかし、ChatGPTと話すたびに、ChatGPTはユーザーが以前に伝えた内容をしっかりと覚えています。

確かに、ChatGPTが以前の会話を覚えているのは、記憶（Memory）機能を使用しているからです。記憶機能により、以前の対話のコンテキストが記録され、そのコンテキストが最新の呼び出し時にモデルに提示されます。チャットボットの構築において、記憶機能は非常に重要です。

第八章　メモリ

➢ ConversationChainについて

ただし、LangChainにおける記憶機能の具体的な実装について説明する前に、まずConversationChainについて見てみましょう。このChainの主な特徴は、AIのプレフィックスと人間のプレフィックスを含む対話要約フォーマットを提供することです。この対話フォーマットは、記憶機能と非常に密接に統合されています。

簡単な例を見て、ConversationChainに内蔵されたプロンプトテンプレートを表示してみると、この対話フォーマットの意味が理解できるでしょう。

```
from langchain import OpenAI
from langchain.chains import ConversationChain

llm = OpenAI(
    temperature=0.5,
    model_name="gpt-3.5-turbo-instruct"
)

conv_chain = ConversationChain(llm=llm)

print(conv_chain.prompt.template)
```

第八章　メモリ

- ConversationChainについて

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

Current conversation:
{history}
Human: {input}
AI:
```

ここでのプロンプトは、人間（ユーザー ）と人工知能（text-davinci-003）との間の基本的な対話フレームワークを設定しています。具体的には、次のような内容です：**「以下は、人間とAIとの友好的な対話です。AIはおしゃべりであり、コンテキストからの多くの具体的な詳細を提供します。」** (The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context.)

第八章　メモリ

- ConversationChainについて

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

Current conversation:
{history}
Human: {input}
AI:
```

また、このプロンプトは以下の内容を説明することで、AIの幻覚（不正確な情報の生成）を減らそうとしています：**「AIが質問の答えを知らない場合は、正直に知らないと答えます。」**（If the AI does not know the answer to a question, it truthfully says it does not know.）

第八章　メモリ

- ConversationChainについて

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

Current conversation:
{history}
Human: {input}
AI:
```

その後、二つのパラメータ **{history}** と **{input}** が見られます。

{history} は会話の記憶を保存する場所であり、つまり人間と人工知能の間の対話の履歴情報です。

{input} は新しい入力の場所で、AIとの対話中にテキストボックスに入力する内容と考えてください。

これらの二つのパラメータはプロンプトテンプレートを通じてLLMに渡され、返される出力は対話の継続となることを期待しています。

第八章　メモリ

- ConversationChainについて

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

Current conversation:
{history}
Human: {input}
AI:
```

そうすると、**{history}** パラメータと Human および AI というニつのプレフィックスがあれば、対話の履歴情報をプロンプトテンプレートに保存し、それを新しい対話の際にモデルに渡すことができます。これが記憶機能の基本的な原理です。

第八章　メモリ

➢ ConversationBufferMemoryを使用する

LangChainでは、ConversationBufferMemory（バッファ記憶）を使用することで、最もシンプルな記憶機能を実現できます。

次に、ConversationChainにConversationBufferMemoryを導入してみましょう。

（以下のコードでは、import文や初期化の部分が省略されています。）

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

llm = OpenAI(temperature=0.5, model_name="gpt-3.5-turbo-instruct")

conversation = ConversationChain(llm=llm, memory=ConversationBufferMemory() )

conversation("お姉さんの誕生日が明日ですね。誕生日の花束を用意する必要があります。")
```

> 「お姉さんが明日誕生日なんですね、素晴らしいですね！誕生日の花束をいくつかおすすめできますよ。どんな花束をお望みですか？バラやカーネーション、チューリップなど、たくさんの種類があります。」

第八章　メモリ

➢ ConversationBufferMemoryを使用する

次回の対話では、これらの記憶がプロンプトの一部として渡されます。

```
conversation("彼女はピンクが好きです。")
print(conversation.memory.buffer)

>Human:お姉さんの誕生日が明日ですね。誕生日の花束を用意する必要があります。
AI: 「お姉さんが明日誕生日なんですね、素晴らしいですね！誕生日の花束をいくつかおすすめできますよ。どんな花束をお望みですか？バラやカーネーション、チューリップなど、たくさんの種類があります。」
Human:彼女はピンクが好きです。
AI:  「では、誕生日にピンクのバラの花束をお勧めします。」
```

実際には、これらのチャット履歴情報は、ConversationChainのプロンプトテンプレート内の {history} パラメータに渡され、新しいプロンプト入力がチャット記録を含む形で構築されます。記憶機能を利用することで、LLMは以前の対話内容を把握することができますが、すべての内容を単純に保存することでLLMに最大限の情報を提供できます。ただし、新しい入力には多くのトークンが含まれるため、応答時間が遅くなり、コストが増加する可能性があります。また、LLMのトークン制限（コンテキストウィンドウ）があるため、対話が長すぎると記憶できなくなります（text-davinci-003やgpt-3.5-turboでは、1回の最大入力制限は4096トークンです）。

第八章　メモリ

➢ ConversationBufferWindowMemoryを使用する

次に、トークンが多すぎる場合やチャット履歴が長すぎる場合のいくつかの解決策を見てみましょう。

ConversationBufferWindowMemoryは、バッファウィンドウメモリであり、最新の数回の人間とAIのインタラクションだけを保存するという考え方に基づいています。そのため、以前の「バッファメモリ」にウィンドウ値 k を追加した形になります。つまり、過去のインタラクションのうち、一定の数だけを保持し、それ以前のインタラクションは「忘れる」ことになります。

```
llm=OpenAI(temperature=0.5,  model_name="gpt-3.5-turbo-instruct")

conversation=ConversationChain(llm=llm,
                               memory=ConversationBufferWindowMemory(k=1))
```

指定された例では、**k=1** に設定されており、これによりウィンドウはAIとの最新のインタラクションのみを記憶します。つまり、直前の人間の応答とAIの応答のみを保持することになります。この方法は最近のインタラクションの記憶には適していますが、トークンの使用量を制限します。長期と最近のインタラクションを混在させる必要がある場合は、他の方法も検討する必要があります。

第八章　メモリ

➢ ConversationSummaryMemoryを使用する

ConversationSummaryMemoryの概念は、対話の履歴を要約し、その要約を {history} パラメータに渡すというものです。この方法は、過去の対話を要約することで、トークンの過剰使用を回避することを目的としています。

ConversationSummaryMemory には以下の主要な特徴があります。

対話の要約：この方法では、対話履歴全体を保存するのではなく、新しい対話が発生するたびにその内容を要約し、それを以前のすべての対話の「実行要約」に追加します。

LLMを使用した要約：この要約機能は、別のLLM（大規模言語モデル）によって行われます。つまり、対話の要約は実際にAI自体が処理するものです。

長時間の対話に適している：この方法は、長時間にわたる対話に特に効果を発揮します。初期の段階では多くのトークンが使用されますが、対話が進むにつれて、要約方法のトークン使用量は抑制されます。一方で、通常のバッファーメモリモデルでは、トークンの使用量が対話の長さに比例して直線的に増加します。

第八章　メモリ

➢ ConversationSummaryMemoryを使用する

次に、ConversationSummaryMemoryを使用したコード例を見てみましょう。

```
from langchain.chains.conversation.memory import ConversationSummaryMemory


conversation = ConversationChain(
    llm=llm,
    memory=ConversationSummaryMemory(llm=llm)
)
```

ConversationSummaryMemoryを使用する場合、ここでの **{history}** は、人間とAIの以前の対話を単純にコピー＆ペーストしたものではなく、要約・整理された総括的な情報となります。ここでは、開発者はLLMを単に各ラウンドの質問に回答させるだけでなく、過去の対話を要約するためにもLLMを活用し、トークンの消費量を節約しています。

第八章　メモリ

➢ ConversationSummaryMemoryを使用する

ConversationSummaryMemoryの利点は、長い対話においてトークンの使用量を減らすことができ、その結果、より多くの対話内容を記録できる点にあります。また、使用も直感的でわかりやすいです。しかし、短い対話においては、逆にトークンの使用量が増える可能性があります。さらに、対話の履歴は中間要約を行うLLMの能力に完全に依存しており、そのためにトークンを使用する必要があり、コストが増加するだけでなく、対話の長さを制限するものではありません。

対話履歴の要約によってトークンの使用を最適化・管理することで、ConversationSummaryMemoryは、複数回にわたる長時間の対話が想定されるシナリオに適した手法を提供します。しかし、この方法もトークン数の制限を受けており、一定の時間が経過すると、依然として大規模モデルのコンテキストウィンドウの制限を超えることになります。

また、要約の過程では、直近の対話と過去の対話が区別されておらず（一般的に、直近の対話の方が重要です）、そのため新しい記憶管理方法の模索が今後も必要です。

第八章　メモリ

➢ ConversationSummaryBufferMemoryを使用する

最後にご紹介する記憶メカニズムは **ConversationSummaryBufferMemory**、すなわち対話要約バッファメモリです。これは、ConversationSummaryMemory や ConversationBufferWindowMemory などの様々な記憶メカニズムの特徴を組み合わせたハイブリッドメモリモデルです。このモデルは、対話中に初期のやり取りを要約しながら、できるだけ最近のやり取りの原始的な内容を保持することを目的としています。

これは、max_token_limit パラメータによって実現されています。最新の対話のテキストの長さが300字以内であれば、LangChain は原始の対話内容を記憶します。対話のテキストがこのパラメータの長さを超える場合、モデルは設定された長さを超えた内容を要約して、トークンの使用量を節約します。

第八章　メモリ

➢ ConversationSummaryBufferMemoryを使用する

次に、ConversationSummaryBufferMemoryを使用したコード例を見てみましょう。

```
from langchain.chains.conversation.memory import
ConversationSummaryBufferMemory

conversation = ConversationChain(
    llm=llm,
    memory=ConversationSummaryBufferMemory(
        llm=llm,
        max_token_limit=300)
)
```

第八章　メモリ

➢ ConversationSummaryBufferMemoryを使用する

ConversationSummaryBufferMemory の利点は、要約を通じて初期のやり取りを思い出すことができ、バッファーによって最近のやり取りの情報が失われないようにする点です。ただし、短い対話においては、ConversationSummaryBufferMemoryもトークンの使用量を増加させることがあります。

総じて、ConversationSummaryBufferMemory は多くの柔軟性を提供します。これまでの記憶タイプの中で、初期のやり取りを思い出しながら最近のやり取りも完全に保存できる唯一の方法です。トークンの節約という点でも、ConversationSummaryBufferMemory は他の方法と比べて競争力があります。

第八章　メモリ

➢ まとめ

今日は、1つの対話チェーンと4つの対話記憶メカニズムについてご紹介しましたので、これら4種類の記憶を総合的に比較するために、表にまとめてみましょう。

記憶メカニズム	長所	短所
ConversationBufferMemory	1.LLM に最大量の情報を提供する。 2. 方法がシンプルで直感的。	1. より多くのトークンを使用するため、応答時間の増加やコストが高くなる。 2. 長い会話ではトークン制限を超える可能性がある。
ConversationBufferWindowMemory	1. 最近のやり取りのみを保持し、トークンの使用量が少ない。 2. ウィンドウサイズが調整可能で柔軟性が高い。	1. 初期のやり取りを記憶できない。 2. ウィンドウが大きすぎるとトークンの使用量が増える可能性がある。
ConversationSummaryMemory	1. 長い会話においてトークン使用量を減らすことができる。 2. より長い時間の会話を許可する。 3. 比較的直接的で直感的な実装。	1. 短い会話ではトークン使用量が増える可能性がある。 2. 会話の記憶は LLM の集約能力に依存する。
ConversationSummaryBufferMemory	1. 初期のやり取りも思い出すことができる。 2. 最近の情報を見逃さない。 3. 柔軟性が高い。	短い会話では要約によりトークンの量が増える可能性がある。

第八章　メモリ

➢　まとめ

また、対話の輪が増えるにつれて、各記憶メカニズムがトークンをどの程度消費するかを示した図もあります。意図としては、ConversationSummaryBufferMemory や ConversationSummaryMemory などの記憶メカニズムは、対話輪数が少ない時にはトークンがやや無駄になる可能性がありますが、多くのラウンドが進むにつれてトークンの節約効果が次第に現れるということです。一方、ConversationBufferWindowMemory はトークンの節約が最も直接的ですが、Kラウンド以前の対話内容を完全に忘れてしまうため、特定のシナリオには最適ではないこともあります。