Chains in LangChain

1. Introduction to Chains

Chains are a core concept in LangChain, designed to streamline the development of Large Language Model (LLM) applications by enabling the creation of "pipelines" that connect multiple smaller steps.

- **Necessity of Chains:** Developing LLM-based applications typically involves "multiple smaller steps" (e.g., prompt design, sending input to LLM, processing LLM response). Manually executing each of these steps can be "very manual," especially for complex applications. Chains automate this process.
- **Pipelining:** Chains allow you to "connect these three steps and create a pipeline." The key benefit is that the "output of your first step automatically acts as the input for the second step, and the output of the second step acts as the input for the third step, and so on."
- **Automation:** Once a pipeline (chain) is built, you only need to "provide input at the first step and trigger this pipeline." The entire sequence of operations will then execute automatically, leading to the final output without manual intervention. This "makes your work much easier."
- **Versatility:** Chains are not limited to linear or sequential pipelines. They enable the creation of "different structures of pipelines," including:
- **Parallel Processing:** "You can actually perform parallel processing...you can create parallel chains."
- **Conditional Chains:** "Based on a condition, you can create conditional chains," allowing different chains to execute based on specific conditions. This versatility makes it possible to "build very complex applications."

2. Core Components and Workflow

Prior to Chains, the LangChain playlist covered:

- **Models:** Interacting with different AI models.
- **Prompts:** Various ways to send inputs to LLMs, including structured output generation and the use of Output Parsers.

Chains bring these components together. The video introduces the **LangChain Expression Language (LCEL)**, a "declarative syntax" using the pipe operator (|) to connect components, making chain formation intuitive (e.g., prompt | model | parser).

3. Types of Chains

Generally there is Three key types of chains:

A. Simple Sequential Chain:

- **Purpose:** To illustrate the basic concept of connecting sequential steps.
- **Example:** A three-step chain:

1. User prompt input (e.g., a "topic").
2. Sending the prompt to an LLM.
3. Displaying the LLM's response.

- **Implementation:**PromptTemplate to define the prompt (e.g., "Generate Five Interesting Facts About {topic}").
- ChatOpenAI as the LLM.

- StringOutputParser to extract the content from the LLM's response.
- Chain creation: prompt | model | parser
- **Benefit:** Simplifies the execution of multiple steps into a "single line," compared to manually invoking each component.
- **Visualization:** Chains can be visualized using chain.get_graph().print_ascii(), which provides a clear ASCII representation of the chain's flow.

B. Long Sequential Chain (Multi-LLM Interaction):

- **Purpose:** To demonstrate a more complex sequential workflow involving multiple LLM calls.
- **Example:** An application that takes a user-provided topic (e.g., "Unemployment in India"), generates a detailed report using an LLM, and then feeds that report back into the *same* LLM (with a second prompt) to extract a five-point summary.
- **Implementation:**Two PromptTemplate instances:
- prompt_one: "Generate a detailed report on {topic}."
- prompt_two: "Generate a five-pointer summary from the following text:\n{text}."
- ChatOpenAI as the model.
- StringOutputParser as the parser.
- Chain creation: prompt_one | model | parser | prompt_two | model | parser
- **Key Takeaway:** Demonstrates how the output of one LLM call, after parsing, can become the input for a subsequent prompt and LLM call within the same chain, allowing for complex multi-step reasoning.

C. Parallel Chain:

- **Purpose:** To execute multiple sub-chains simultaneously, processing the same input in different ways.
- **Example:** An application that takes a detailed text document (e.g., on "Linear Regression") and "parallely" generates two distinct outputs:

1. "Notes" from the document.
2. A "quiz" (five short Q&A) from the document. These two outputs are then combined and displayed to the user.

- **Architecture:**User provides text.
- Text is sent to **two separate models/chains in parallel.**
- Model One generates notes.
- Model Two generates a quiz.
- Both outputs are then sent to a **third model/chain** to merge them.
- **Implementation:**RunnableParallel: A LangChain construct used to execute multiple chains in parallel. It takes a dictionary where keys are output names and values are the sub-chains.
- Separate PromptTemplate instances for notes (prompt_one) and quiz (prompt_two).
- Two different LLM models (ChatOpenAI and ChatAnthropic - Claude 3 was initially attempted but corrected to claude-3-opus-20240229).
- StringOutputParser for both parallel outputs.

- A third PromptTemplate (prompt_three) for merging the notes and quiz, taking notes and quiz as input variables.
- The overall chain: parallel_chain | merge_chain (where parallel_chain contains the RunnableParallel setup and merge_chain combines prompt_three, a model, and the parser).
- **Visualization:** The graph clearly shows two parallel branches before converging into a final merging step.

D. Conditional Chain:

- **Purpose:** To execute different sub-chains based on a specific condition derived from an earlier step's output. This is analogous to an if-else statement.
- **Example:** A feedback sentiment analysis application:

1. User provides feedback text (e.g., "This is a terrible smartphone").
2. An LLM classifies the sentiment as "Positive" or "Negative."
3. Based on the classified sentiment:

- If "Positive," a positive-response chain is executed (e.g., "Thank you for your kind words!").
- If "Negative," a negative-response chain is executed (e.g., "I'm sorry to hear that...").
- **Key Challenge & Solution:** LLM outputs for classification might be inconsistent (e.g., "Positive," "The sentiment is positive," etc.). To ensure reliable branching, the output needs to be "structured."
- **Structured Output:** Achieved using PydanticOutputParser and a Pydantic BaseModel (Feedback class with a Literal sentiment field for "Positive" or "Negative"). This ensures the LLM output for sentiment is *always* either "Positive" or "Negative."
- **Implementation:**PromptTemplate (prompt_one) for sentiment classification, enhanced with format_instructions from PydanticOutputParser.
- ChatOpenAI as the model.
- PydanticOutputParser (parser_two) for structured sentiment output.
- Two additional PromptTemplate instances (prompt_two for positive response, prompt_three for negative response).
- RunnableBranch: The core component for conditional execution. It takes a list of tuples, where each tuple is (condition_lambda_function, chain_to_execute). A final default_chain is provided if no conditions are met.
- **Conditions:** Lambda functions check the .sentiment attribute of the structured output (e.g., lambda x: x.sentiment == "Positive").
- **Chains for Branches:** Simple sequential chains (e.g., prompt_two | model | parser) are used for the positive and negative branches.
- **Default Chain Handling:** A RunnableLambda is used to convert a simple lambda function (e.g., lambda x: "Could not find sentiment") into a Runnable object that can be part of the chain, fulfilling the RunnableBranch requirement.
- The overall chain: classification_chain | branch_chain.
- **Visualization:** The graph shows a single input leading to a "branch" point, from which only one path (positive or negative) is followed.