Learn and Grow

Generative AI LangChain

ASHIMA MALIK

ashimamalik58@gmail.com

CHAINS

Chains are like recipes for your Langchain agent. They describe the step-by-step process the agent follows to complete a task.

Each step in a chain can involve: Interacting with a Langchain tool (e.g., web search, summarization) Performing data manipulation or processing Making decisions based on the information gathered

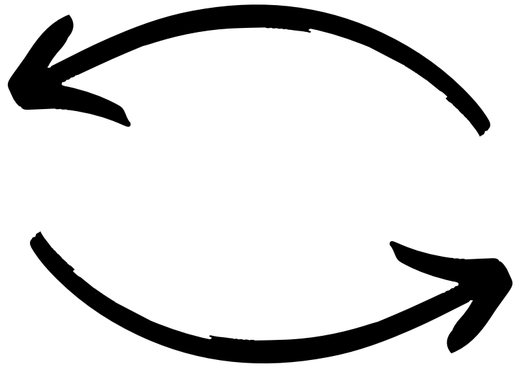
sequence of steps

follows the steps

CHAINS

AGENTS

interpret user input



TASK

execute chain to generate response

ASHIMA MALIK

Chains are like recipes for your Langchain agent. They describe the step-by-step process the agent follows to complete a task.

Simple Sequential Chain

This is the most basic type of chain. It involves a linear sequence of steps, where the output from one step becomes the input for the next. It's ideal for simple tasks with a single input and a single output.

Transform Chain

This type of chain allows you to apply transformations to the data at each step. It's useful when you need to manipulate or process information before feeding it to the next step.

Router Chain

This chain is more complex and allows for conditional branching based on the information gathered at different points. It's suitable for tasks with multiple possible paths or outcomes depending on the user's input or retrieved information.

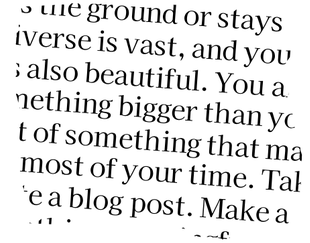
LLMChain

This chain focuses on interactions with large language models (LLMs). It allows you to use LLMs for tasks like text generation, translation, or question answering within your workflow.

ASHIMA MALIK

Summarize the news article?

Use a web search tool to find relevant articles



Extract the text from the articles

Use a summarization tool to generate a summary of each article

Combine the summaries into a final response

ASHIMA MALIK

Simple Sequential Chain

Place?

Best place to visit in {place}?

Prompt

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

1

Cost?

Given a list of places, please estimate the cost to stay there {cost}?

Prompt

```
cost_chain =
LLMChain(llm=llm,
prompt=prompt_t
emplate)
```

2

```
final_chain =
SimpleSequentialChain
(chains=[place_chain,
cost_chain],
verbose=True)
```

ASHIMA MALIK

Sequential Chain

Review **Summary** **Sentiment**

```
template1 = "Give a summary of this product review: \n{review}"
prompt1 = ChatPromptTemplate.from_template(template1)
chain_1 = LLMChain(llm=llm,
                   prompt=prompt1,
                   output_key="review_summary")
```

```
template2 = "Identify the sentiment in this review summary: \n{review_summary}"
prompt2 = ChatPromptTemplate.from_template(template2)
chain_2 = LLMChain(llm=llm,
                   prompt=prompt2,
                   output_key="sentiment")
```

```
seq_chain = SequentialChain(chains=[chain_1,chain_2],
                            input_variables=['review'],
                            output_variables=['review_summary','sentiment'],
                            verbose=True)
```

ASHIMA MALIK

Summarization

Chain Name	Description	Use Case	Parallelizable?
StuffDocumentsChain	Combines all documents into a single prompt for an LLM.	When the total content fits within the LLM's context window and you want all documents processed in a single call.	No
ReduceDocumentsChain	Iteratively reduces documents in chunks by feeding them to an LLM, then combining the results.	When you have a lot of documents exceeding the LLM's context window and want all documents included (can be batched for efficiency).	Yes
MapReduceDocumentsChain	Runs each document through an LLM first, then reduces the generated summaries using ReduceDocumentsChain.	Similar to ReduceDocumentsChain, but performs an initial individual LLM call on each document before reduction.	Yes
RefineDocumentsChain	Generates an initial answer based on the first document, then iteratively refines it using LLMs on remaining documents.	When you want to build an answer sequentially, refining based on previous outputs. Not suitable for parallelization.	No

MEMORY

ASHIMA MALIK

ConversationBufferMemory

- Designed specifically for chatbots and similar applications.
- Stores the history of user queries and chatbot responses within a single conversation
- Enables context-aware responses by allowing the model to access past interactions
- Might have a limited size, prioritizing recent exchanges.

```
llm = OpenAI(temperature=0)
conversation = ConversationChain(
    llm=llm,
    verbose=True,
    memory=ConversationBufferMemory()
)
```

ConversationBufferWindowMemory

- In ConversationBufferWindowMemory, we provide the value of k, which keeps the k no of interactions to be remembered in memory.

```
conversation_with_summary = ConversationChain(
    llm=OpenAI(temperature=0),
    # k=2, to keep the last 2 interactions in memory
    memory=ConversationBufferWindowMemory(k=2),
    verbose=True
)
```

ConversationSummaryMemory

- Focuses on capturing the essence of a conversation rather than storing every detail.
- May summarize key points or user intents.
- Useful for tasks requiring a high-level understanding of the conversation flow.

```
conversation_with_summary = ConversationChain(
    llm=llm,
    memory=ConversationSummaryMemory(llm=OpenAI()),
    verbose=True
)
```

ConversationSummaryBufferMemory

ConversationSummaryBufferMemory appears to be a combination of two existing Langchain memory types:

- **ConversationSummaryMemory**
- **Conversation BufferMemory**

This system remembers recent conversations, but instead of simply discarding older ones, it creates a condensed version (summary) of them and uses both the summary and the latest details for better understanding. It decides when to get rid of older details based on the total number of words spoken, not just how many conversations have happened.

```
conversation_with_summary = ConversationChain(
    llm=llm,
    # set a low max_token_limit for the purposes of testing.
    memory=ConversationSummaryBufferMemory(llm=OpenAI(),
                                           max_token_limit=50),
    verbose=True,
)
```

ConversationTokenBufferMemory

- Similar to Conversation BufferMemory but stores individual tokens (words) instead of complete utterances.
- Provides a finer-grained view of the conversation for models that benefit from analyzing word sequences.
- Might be useful for tasks like sentiment analysis or topic modeling.

```
conversation_with_summary = ConversationChain(
    llm=llm,
    # set a low max_token_limit for testing.
    memory=ConversationTokenBufferMemory(llm=OpenAI(),
        max_token_limit=70),
    verbose=True,
)
```

ConversationKnowledgeGraphMemory

- Represents the conversation as a knowledge graph, where nodes represent entities and edges denote relationships between them.
- Allows the model to reason about the information discussed in the conversation.
- Useful for tasks like question answering or information retrieval based on the conversation context.

```
conversation_with_kg = ConversationChain(
    llm=llm, verbose=True, prompt=prompt,
    memory=ConversationKGMemory(llm=llm)
)
```

VectorStoreRetrieverMemory

- Integrates with external vector stores like Faiss to store and retrieve dense vector representations of data points (e.g., documents, user profiles).
- Enables efficient similarity search based on these vector representations.
- Can be used for tasks like recommendation systems or information retrieval based on semantic similarity.

```
retriever = vectorstore.as_retriever(search_kwargs=dict(k=1))
memory = VectorStoreRetrieverMemory(retriever=retriever)
```

Callbacks

- LangChain provides a callback system that allows you to hook into the various stages of your LLM application. This is useful for logging, monitoring, streaming, and other tasks.

- CallbackHandler acts as a middleman between an application and security services. Its primary function is to handle user interactions required for authentication and authorization processes.